RL-TR-97-134
Final Technical Report
October 1997

CRONUS/MACH INTEGRATION

BBN Systems and Technologies

Susan Pawlowski Sours, James C. Berets,
E. John Sebes and Edward F. Walker

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19980113 033

DTIC QUALITY INSPECTED 3

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-134 has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

FOR THE DIRECTOR: John A. Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3AB, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Oct 97	Final Jun 93 - Mar 96

4. TITLE AND SUBTITLE

CRONUS/MACH INTEGRATION

5. FUNDING NUMBERS

C - F30602-93-C-0033
PE - 63728F
PR - 2530
TA - 01
WU- 57

6. AUTHOR(S)

Susan Pawlowski Sours, James C. Berets, E. John Sebes, and Edward F. Walker

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

BBN Systems and Technologies
10 Moulton St.
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

8149

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3AB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-134

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Thomas F. Lawrence, C3AB, 315-330-2925

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

N/A

13. ABSTRACT *(Maximum 200 words)*

The objective of this effort is to integrate the capabilities of the Cronus distributed computing system with the capabilities of the Mach kernel to achieve superior functionality, performance, and usability. How Mach mechanisms such as Mach IPC and POSIX threads can be integrated into Cronus was investigated. Investigations are summarized as follows: Examination of the viability, benefits, and disadvantages of porting the Cronus communications protocols to the x-kernel framework. Investigation of mechanisms for priority-based manager task scheduling. Investigation of the tradeoffs of using a non-canonical data representation for communicating between client and manager located on the same platform. Exploration of uses of Mach inter-process communication to increase the efficiency of Cronus communication, and to integrate local Cronus/Mach communication, distributed Cronus/Mach communication, and distributed communication between Mach and non-Mach Cronus hosts. Investigation of the potential uses of Mach external paging mechanisms. Examination of the similarities and differences between Cronus and the OSF Distributed Computing Environment and between Cronus and the Object Management Group's CORBA specification. Enhancement of Cronus to support the CORBA specifications. The resulting system known as Corbus supports the CORBA 2.0 specification for C mappings and for C++ mappings including exceptions.

14. SUBJECT TERMS

Distributed Computing Environment, Cronus, quality of service, MACH, Corbus, migrating thread

15. NUMBER OF PAGES

160

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

DTIC QUALITY INSPECTED 3

List of Figures

Part I

Project Summary

This document is the final technical report for the Cronus/Mach Integration Project. The work for the project was performed over the period from June 1993 through March 1996. This project was funded by the U.S. Air Force Rome Laboratory (RL) under contract F30602-93-C-0033.

The objective of this effort is to integrate the capabilities of the Cronus distributed computing system with the capabilities of the Mach kernel to achieve superior functionality, performance and usability. This report was written by BBN together with its subcontractor, Trusted Information Systems (TIS).

Our work focused on investigating how Mach mechanisms can be integrated with Cronus. Our approach to the Cronus/Mach integration is different from the usual approach taken to port Cronus to a new platform. Because Cronus is required to operate on a variety of operating systems, generally the approach taken is to choose implementation mechanisms that offer the most portability to a wide variety of platforms. As a result, a typical port of Cronus to a new UNIX platform requires only minor code changes. However, frequently under this approach, we cannot use the most effective or efficient mechanisms provided by the native operating system. Under this effort, we investigated how Mach mechanisms such as Mach IPC and POSIX threads can be integrated into Cronus in sophisticated ways to produce a new, highly functional distributed environment.

The Mach platform for our investigations is the Mach 3.0 (Microkernel). Its key features include (1) complete multiprocessor support; (2) multiple threads of control within a single address space; (3) interprocess communication; (4) flexible memory management via application-specific "external pagers"; (5) support for multiple operating environment "personalities" (e.g. INIX, MS-DOS).

Our investigations are summarized as follows:

- BBN examined the viability, benefits and disadvantages of porting the Cronus communication protocols to the x-kernel framework. Replacing Cronus's current socket based implementation with an x-kernel based implementation would provide a dramatic increase in Cronus's communication performance on Unix platforms.

- BBN investigated mechanisms for priority based manager task scheduling. A priority ordered thread based implementation would offer improved scheduling of operation invocations over the existing non-preemptive tasking mechanism by ensuring that each operation gets

an equal amount of processor time. Various thread packages were examined, with POSIX pthreads selected as the package offering the most advantages.

- BBN investigated the tradeoffs of using a non-canonical data representation for communicating between client and manager located on the same platform. This approach has the benefit of improved performance when a direct connection between client and manager is made; however, in the case where an invocation is handled by the kernel this benefit is balanced out by increased data transmission overhead.

- TIS explored the use of Mach inter-process communication to increase the efficiency of Cronus communication, and to integrate local Cronus/Mach communication, distributed Cronus/Mach communication, and distributed communication between Mach and non-Mach Cronus hosts.

- TIS investigated the potential uses of Mach external paging mechanisms, with the goal of combining Cronus and Mach functionality to provide distributed typed shared memory mechanisms that are consistent with Cronus's object oriented client server concept of operation.

BBN was requested by Rome Laboratory to investigate two additional distributed computing technologies related to Cronus. These investigations examined two important developments in distributed computing: (1) the Open Software Foundation's Distributed Computing Environmant, which has gained popularity and a small customer base as a distributed environment; (2) the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) specification for communicating between objects on different platforms. The CORBA and OSF/DCE investigations, similar to the Mach investigations described above, examine ways in which Cronus can be integrated with a commercially available technology.

- BBN examined the similarities and differences between Cronus and the OSF Distributed Computing Environment, which are two functionally similar distributed computing environments.

- BBN examined the similarities and differences between Cronus and the Object Management Group's CORBA specification.

The reports covering each of these areas of investigation have been previously reported and submitted as part of the Cronus/Mach Integration Interim Technical Report for this activity. It is available as BBN Report No. 8089, February 27, 1995.

As a result of these earlier investigations BBN suggested and Rome Laboratory agreed that the focus of the remaining effort be on enhancing Cronus to support the CORBA specifications of the Object Management Group. The resulting system, known as Corbus, supports the CORBA 2.0 specification for C mappings and for C++ mappings including exceptions. Corbus is described in the Corbus System/Segment Specification, SOftware Design Document, User's Manual, Operator's Manual, Tutorial, and Release Notes all of which were separately delivered as part of this effort.

As a complementary activity to the CORBAization of Cronus, TIS investigated the issues involved with the integration of Corbus with the Mach microkernel. The result of this investigation is reported as section 2 of this final report.

Part II

Corbus and Mach: Advanced System Platforms for Distributed Object Computing

Chapter 1

Introduction

Corbus is a CORBA-compliant Object Request Broker (ORB), a software system which provides an infrastructure for the execution of distributed object applications. CORBA is the Common Object Request Broker Architecture, defined by the Object Management Group (OMG), a consortium of operating system (O/S) and application vendors. OMG is defining standards for distributed object application interoperability based on CORBA. In addition to the primary ORB function, Corbus provides additional features that are derived from its history as an advanced technology development vehicle. It has been used for distributed client/server object-oriented applications. One such feature is a set of tools that includes software generation tools which embed application-specific, object-management software in a generalized server framework. The server framework includes platform-independent subsystems for distributed object naming, data replication, and data storage and management.

Because of its advanced features and its nature as a mature base for research, Corbus can combine two complementary roles in the future. First, Corbus can be a vehicle for continuing integration of advanced system features into distributing computing. Second, Corbus can provide a basis for distributed computing in government/military settings that have requirements that are not met by commercial ORBs. In these settings, distribution requirements can be met by Corbus, with environment-specific extensions and/or integration of other advanced capabilities.

Mach is a microkernel base for O/Ss. It is currently the subject of research at a variety of organizations, as well as the basis of several emerging commercial O/Ss. Besides being at the heart of several research and com-

mercial O/Ss, Mach is also a platform for the development and deployment of systems with a variety of advanced capabilities. Corbus, by being hosted on a Mach-based system, can benefit from a wide range of Mach-based technologies via the advanced capabilities of such technologies. By utilizing such capabilities, a Corbus/Mach system can provide distributed object computing along with the benefits of these advanced technologies.

This report describes a number of such advanced system capabilities, and how Corbus can leverage off of them to provide their benefits to distributed applications. For a few of these capabilities—such as the mega-multiprocessing of Mach SMP machines—Corbus can provide application-level benefits with little more required than hosting Corbus on the Mach-based platform. In other cases, Corbus internals must make use of the capabilities of the underlying Mach system, either of the Mach kernel or of software based on it.

The next section presents the overall Corbus/Mach architecture and subsequent sections describe one area of advanced system capability, its benefits to distributed object applications, and how Corbus can utilize specific features to provide new capabilities to applications.

Chapter 2

Architecture

Direct use of Mach functionality is the keystone of the architecture of a Mach system for Corbus. This section describes the high-level functional requirements for Corbus, and how they can be satisfied in a way that allows Corbus-based, CORBA-compliant applications to benefit from advanced system capabilities deployed on Mach systems. This section provides background for and a description of the basic Corbus/Mach approach, and describes the primary functional requirements of Corbus and how those requirements are met in a Mach system. In addition, the architecture and interactions of Corbus/Mach components are described, and additional detail about each component is provided. Taken together, these descriptions provide an informal high-level view of the following areas of documentation for a potential Corbus/Mach development: system concept, system requirements, system architecture and top-level component decomposition, inter-component interface definition, and per-component requirements.

2.1 Background

Mach is an O/S microkernel that originated in research at Carnegie Mellon University (CMU). One of Mach's primary advantages is the minimality of the services of the microkernel: process management, virtual memory, interprocess communication (IPC), and devices. Only these services are implemented by supervisor-mode software, and hardware-dependencies are isolated to specific microkernel subsystems. All other system functions may be implemented in user-mode processes, with several resulting benefits of portability, maintainability, and flexibility. User-mode O/S software can

utilize low-level microkernel features, rather than having to implement them as an integral part of high-level system features. As a result, Mach has been used as the basis for a variety of O/S implementations (including Unix, DOS, and Macintosh O/S). It continues to serve as the basis for further development of commercial O/Ss by major vendors.

The most common architecture for Mach-based systems is for application software for a given O/S, e.g., Unix, to execute on a Mach-based system solely by interacting with the user-mode O/S software, e.g., a Unix server. A Unix server is a single Mach task[1] which implements all Unix system features for Unix application software running in Mach tasks. Application software is unaware of the existence of the Mach microkernel. As a result, Unix software can run unchanged with a Unix server on Mach, and requires only the addition of library software to direct Unix system calls to the Unix Server. However, applications cannot make direct use of Mach features.

Therefore, there is a tradeoff at the application level between two goals: (1) application-level usage of Mach features, with the consequence that the application is dependent on Mach; and (2) having a Unix application that runs on both Mach and non-Mach systems. As a result, for Unix and other other O/S personalities,[2] applications are completely reliant on an O/S personality. This approach meets the second goal, but places complete responsibility on the O/S personality for utilization of Mach features. Unfortunately, this has a negative impact on the utilization of advanced system capabilities since such utilization must be done by modifying the O/S software, albeit O/S software that runs in Mach tasks.

[1]The term *task* is used to refer to Mach's process-like abstraction. Mach tasks differ from the proceses of other O/Ss primarily in that tasks are inherently multi-threaded, and in that threads, memory, and IPC capabilities are first-class kernel abstractions just as tasks are.

[2]The term *O/S personality* is used to describe the implementation of an O/S interface on a Mach system. An O/S personality may be implemented as a single server that implements O/S-specific services (as in OSF/1), or as multiple O/S-specific servers, or as multiple "personality-neutral" servers. Also, there may be some element of "O/S emulation" by embedding a significant amount of O/S functionality in library software used by every process. The term O/S personality abstracts away from these details when it is not important whether a personality server, or emulation, or some combination is used.

2.2 Approach

Corbus, as application middleware, is in a position to take a different stance on the tradeoffs identified above. Corbus/Mach is enabled by the basic nature of Corbus as an ORB. That is, application-level software uses the facilities of an underlying O/S to implement general, distributed system-wide services of object-oriented communication. As is usual with ORBs, application-level software provides O/S-independent services in support of object-oriented client/server distributed applications. These services are O/S-independent in the sense that one ORB may have several implementations on different O/S bases, but all would present the same ORB interface to applications.

Corbus/Mach follows similar principles. That is, it can use underlying O/S features. In the case of Mach, the underlying O/S features include the functionality of the Mach microkernel and other advanced system software deployed on it. Like other ORBs, Corbus has O/S-dependent portions of the implementation, and Corbus/Mach includes the use of Mach features. Also, Corbus/Mach presents the same CORBA interfaces to applications, thus shielding applications from dependence on Mach, and allowing applications to benefit from ORB usage of Mach features.

As a result of this strategy, Corbus avoids the trade-offs described above. Mach and Mach-based advanced capabilities can be used *transparently* by middleware-level ORB software. Neither Mach-based O/S personality software nor applications need to be modified in order to take advantage of these capabilities.

Having established this Corbus/Mach strategy, the architectural goal is to meet Corbus functional requirements while minimizing dependence on O/S personalities. By minimizing such dependence, the Corbus/Mach architecture removes O/S servers from the client/server dataflow path. As a result, the client-server dataflow path includes only Corbus, the Mach microkernel, and other support software implemented on the microkernel.

O/S heterogeneity is another factor relevant to the use by Corbus/Mach of services of an O/S personality. If Corbus/Mach's use of Mach features is tightly intertwined with use of additional features of an O/S personality, then the Mach benefits to Corbus would not generally apply to Mach systems, but would instead be specific to a particular Mach system including a particular O/S personality. Isolation of O/S dependencies, to the greatest extent possible, is needed to ensure the broad applicability of Corbus/Mach results.

2.3 Requirements

Of the Corbus/Mach system base issues described in this section, most issues stem from the functionality that Corbus requires from a base system in order for the base system to provide a complete Corbus operating environment. The requirements are grouped into broad areas, according to which component of a Corbus environment to which they apply.

There are three major components:

Core: the Corbus ORB Core, which implements a client/server communication infrastructure. The Core is embodied primarily in a separate process which acts as a message switch that receives object requests from clients; locates objects; sends object requests to object managers; receives object service replies from managers; and sends replies to clients.

Client: a program that makes object requests. It consists of three basic parts: client application-specific software that invokes operations on objects; the application-specific client stub library which implements each operation invocation by sending a request message and receiving a reply message; and the Corbus client communication library (CCL), which implements object requests and replies in terms of general communication protocols.

Manager: a program that implements object requests. It includes Corbus communication library software as well as server-side stubs and application-specific software that implement the operations on the objects it manages.

The following list describes the various areas of requirements, which of the three major components has the requirement, and whether the required functionality can be implemented by the Mach microkernel or some other component.

Process support is needed to run Corbus' operating environment components (Core, clients, and managers). The Mach kernel's task/thread subsystem provides process support, which O/S personalities directly use to implement processes. That is, Mach provides process support in a manner independent of O/S personalities.

Inter-Process Communication (IPC) is needed to facilitate communication between the Core, managers, and clients on the same host. The

Mach microkernel provides a flexible IPC service that is suitable for Corbus. O/S personalities also provide their own IPC mechanisms. In some personality implementations, IPC is implemented via Mach directly between the communicating processes, while in other instances, IPC uses more indirect mechanisms implemented by O/S personality software. For Corbus/Mach, Mach IPC can be used for Corbus communication within one host. As a result, there is no need to use an O/S personality-specific IPC mechanism.

Networking Facilities and Protocols are needed to provide communication between the Core, managers, and clients on different hosts. The Mach microkernel does not provide networking, though O/S personalities, e.g., Unix, do provide it. In order for Corbus/Mach to avoid dependence on a particular O/S personality for networking, communication protocol service must be obtained from an independent protocol server that is not integral to an O/S personality. The University of Arizona's x-kernel has been used as a protocol server on Mach and is a promising candidate for Corbus/Mach. The x-kernel's role in the Corbus/Mach architecture is described below.

File Service is needed by the Core for stable storage of a few types of configuration data; it is also needed by Corbus managers to store object data. Clients do not require file service as part of Corbus functionality, though a client program is free to use files for other purposes. The Mach microkernel does not provide file service, but O/S personalities do. In order for Corbus/Mach to avoid dependence on a particular O/S personality for storage, the Core and managers must obtain file service in one of two ways: either from an independent file server that is not an integral part of an O/S personality; or via an O/S-independent interface to the file services of various O/S personalities.

Client Environment is the term used to describe the broad array of services required by a Corbus client, which can only be met by the full services of an O/S personality. Essentially, a Corbus client is an application program for a particular O/S, but it incorporates Corbus libraries in order to make object invocations. However, clients generally do much more than make object invocations. Clients implement a user interface and rely on the O/S to support it. The O/S typically starts a user's login session; launches application software, e.g., a client program; provides programs with access to devices, e.g., mon-

Requirement:	of Corbus component	Met by:
Process	Core, clients, managers	Mach microkernel
IPC	Core, clients, managers	Mach microkernel
Network	Core, clients, managers	x-kernel
File	Core, managers	file server
File	clients	O/S personality
Other	clients	O/S personality

Figure 2.1: Corbus Requirements Mapping to Mach Components

itor, keyboard, and mouse; and implements a implements a terminal display or bit-mapped graphic display using theses devices, etc. In addition, client programs may make use of other O/S features, e.g., stable storage of user interface preferences.

The following table summarizes the allocation of these requirements from Corbus components to other components of a Corbus/Mach system.

Note that although client programs are dependent on O/S services, the Corbus client communication library is not, because it can obtain IPC and network service independent of O/S personalities. In other words, the client-side part of Corbus software can be independent of O/S personalities, even though clients as a whole may or may not be, depending on the way in which the non-Corbus part of the client is implemented.

2.4 Corbus/Mach Architecture

This section describes a progression of Corbus/Mach architectures based on the foregoing requirements analysis. Figures 2.2, 2.3, and 2.4 illustrate three distinct Corbus architectures. Figure 2.2 shows the standard Corbus configuration on a native kernelized O/S. Figure 2.3 illustrates a similar situation wherein an O/S personality server on Mach provides the same services as a monolithic O/S kernel. Unix is used here as an example because it is a significant Corbus O/S personality. With a native Unix system, the Core, clients, and managers are all Unix processes and use Unix IPC. Both clients and managers can communicate with the Core, which uses network protocols to send/receive messages to/from remote hosts' Cores. The Core and managers use Unix files to store data and clients use other

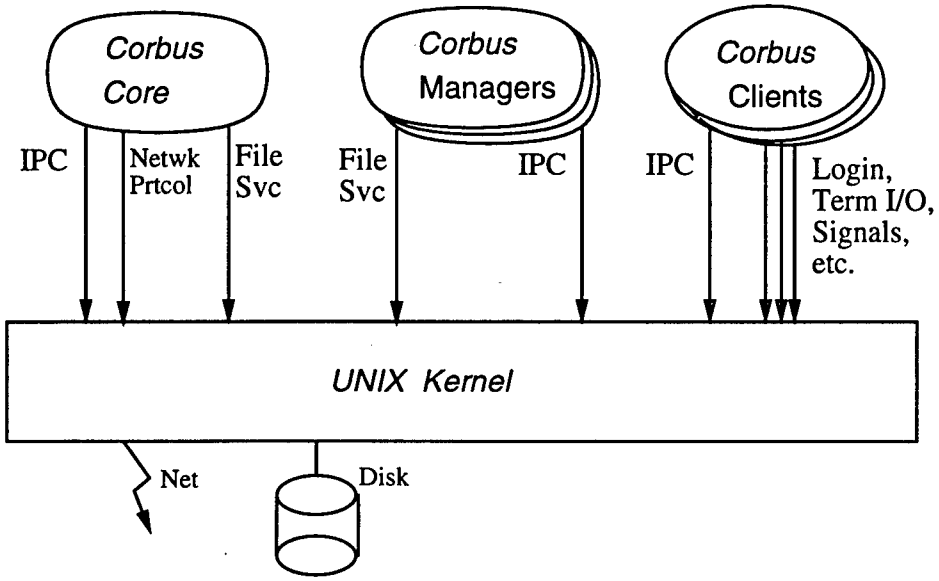

Figure 2.2: Corbus on Native Unix

Unix services to provide user interface and other functionality. The same arrangements apply to the Unix server on Mach, except that the Core, clients, and managers are all Mach tasks, which interact with the Unix server via a Mach kernel mechanism to redirect system calls to the Unix server. The Unix server interacts with the Mach kernel to obtain resources (including network and file devices) which it uses to implement O/S features.

Figure 2.4 illustrates a departure from the pure Unix server approach, in that Corbus components use Mach features as well as Unix features. Figures 2.4, 2.5, and 2.6 show a progression from this point of departure. In Figure 2.4, local Corbus communication is implemented not by Unix IPC, but by Mach's message-based IPC. As a result, Corbus components communicate directly with one another rather than via the Unix IPC. Besides eliminating one O/S dependency from Corbus, this approach notably shortens the client-server communication path. Without using Mach IPC, all Corbus client/server communication must go via *both* the the Unix server and the Corbus Core, resulting in eight hops: from client to Unix-server to Core to Unix-server to manager to Unix-server to Core to Unix-server to

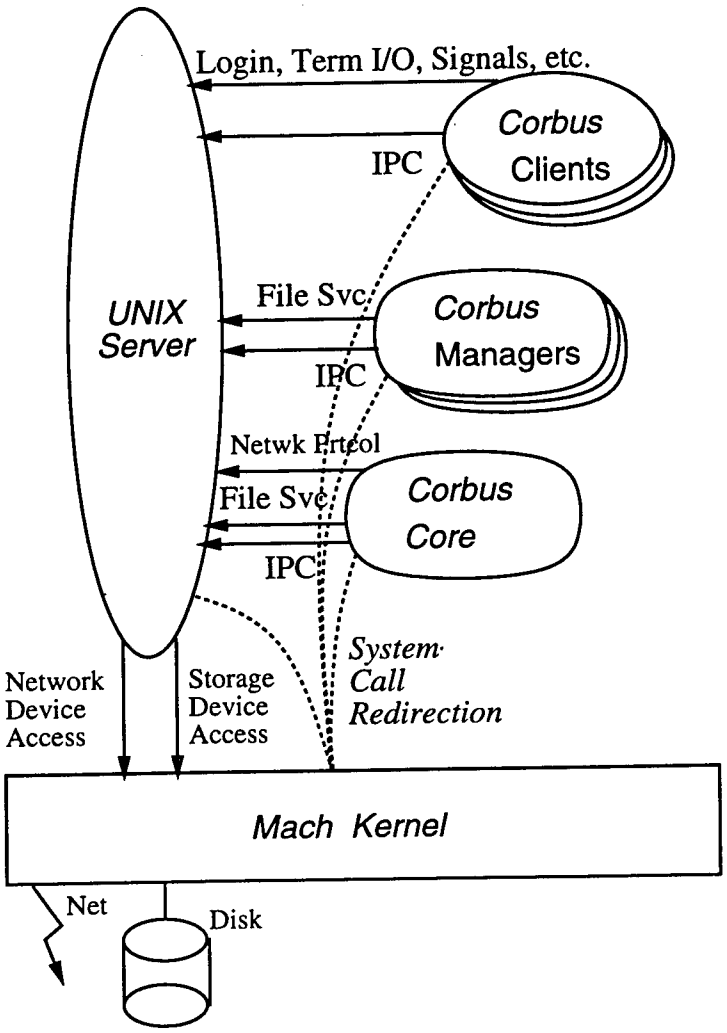

Figure 2.3: Corbus on Unix Server on Mach

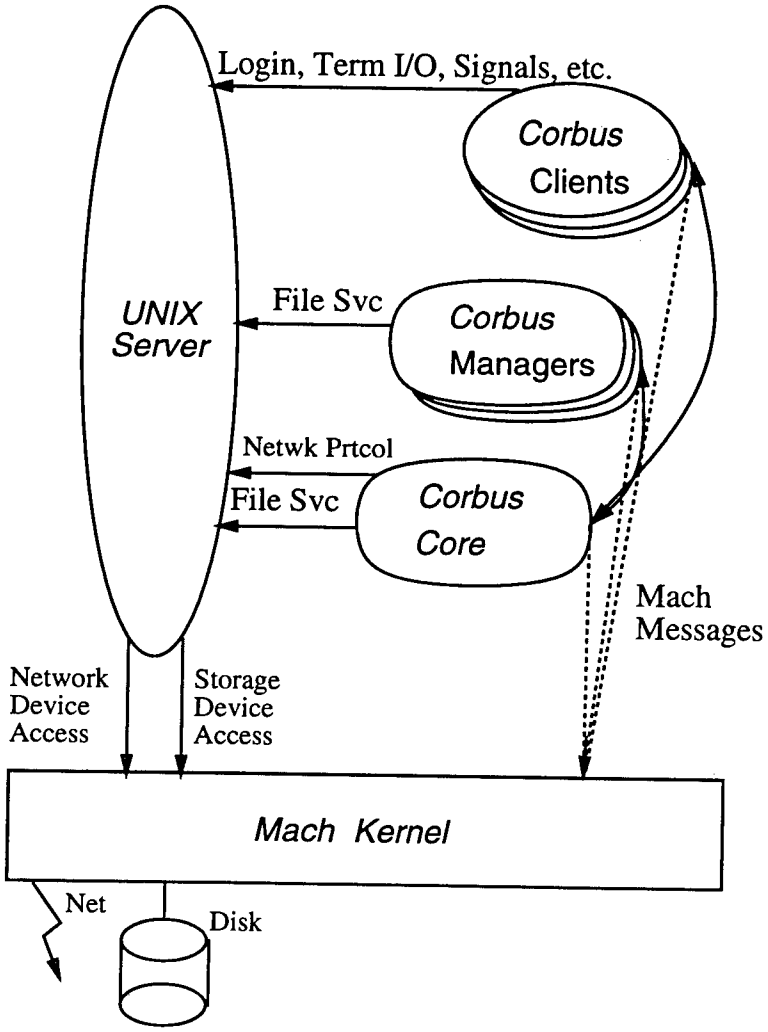

Figure 2.4: Corbus on Mach and Unix Server

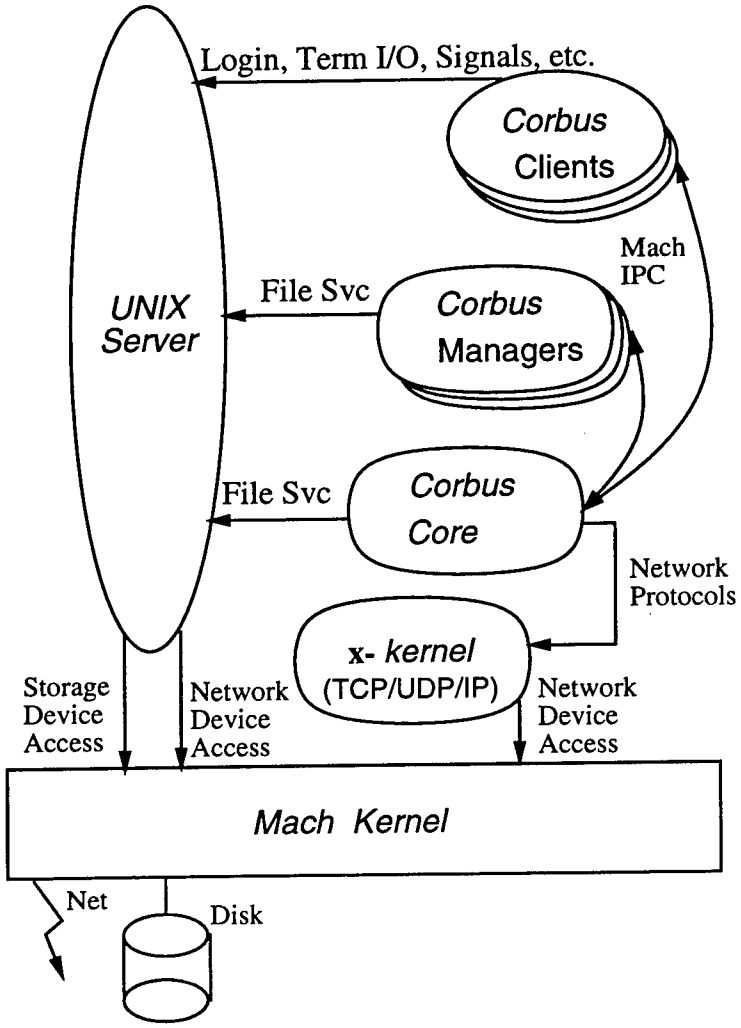

Figure 2.5: Corbus/Mach with x-kernel

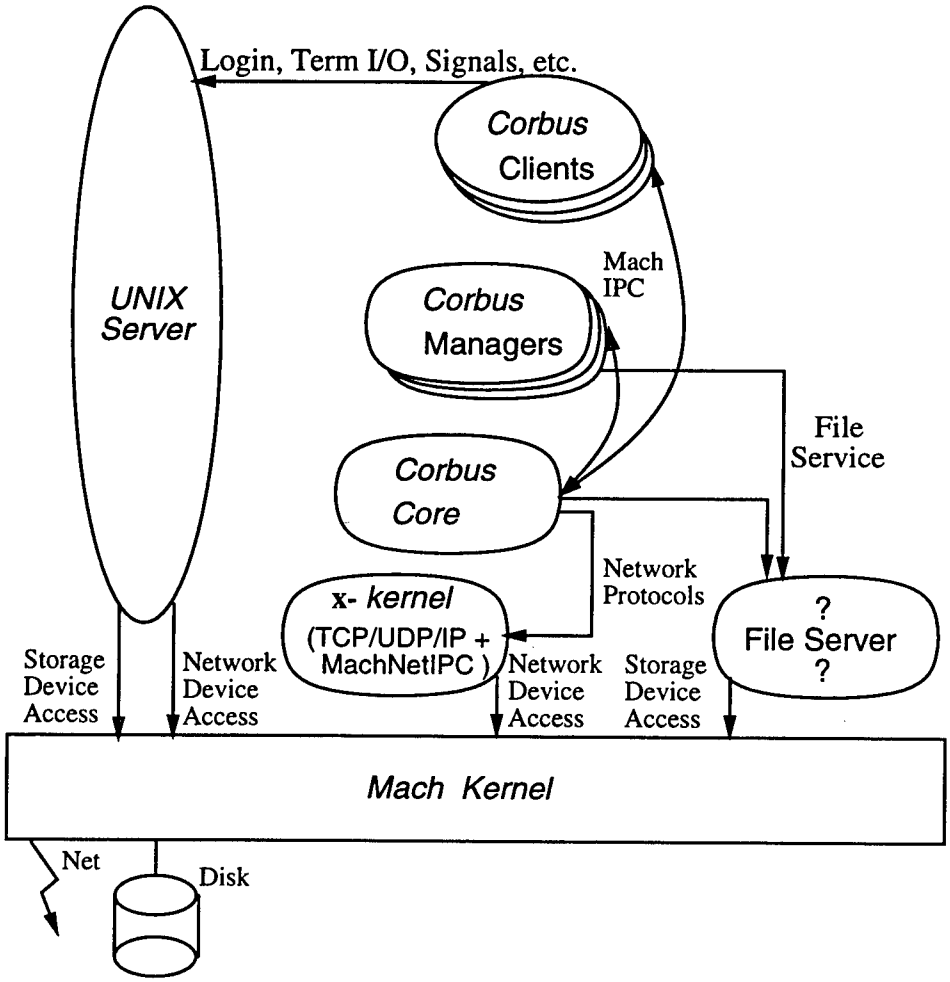

Figure 2.6: Corbus/Mach with File Server

client. With Mach IPC, the Unix server is removed from this dataflow path, and the number of hops is reduced by half.

In Figure 2.5, Corbus uses the x-kernel as a protocol server, thus eliminating the dependency on an O/S personality for network service. The x-kernel uses Mach kernel service to access a network device. The Core uses Mach local IPC to communicate with the x-kernel to obtain Internet Protocol (IP) service, which is used to communicate with the Core on other hosts. The final progression is shown in Figure 2.6, where the Core and managers use an independent file server rather than the file service of the Unix server. As a result, there is no O/S dependency of the Core, or the client communication library, or the application-independent parts of Corbus managers.

As a result of this progression of architectures, there is a different overall dataflow between Corbus/Mach and other Mach software. Figures 2.7 and 2.8 illustrate these differences. Figure 2.7 shows the dataflows between Corbus components in a Unix-only architecture, while Figure 2.8 shows the dataflows that result from the use Mach or Mach-based capabilities for IPC, networking, and files.

2.5 Components

This section discusses the various software components that may be used as part of the Corbus/Mach system base

2.5.1 Mach Microkernel

The Mach microkernel provides the mechanisms for running software in processes. It implements threads of execution in tasks which are composed of virtual memory mappings, intertask communication capabilities, and capabilities for device access. O/S personality software, the Core, clients, and managers all run in Mach tasks. In addition, O/S personalities may provide O/S-specific process management functionality implement in terms of control of tasks. For example, Unix client processes would be part of a process group that could be killed when a user's login session ends. Corbus does not require any additional O/S-specific process management for Core, client, or manager Corbus functionality. The Mach microkernel provides the IPC mechanism for local communication among the Core, clients, and managers.

In addition, the Mach microkernel provides access to devices upon which other software builds high-lever service. O/S personalities and the x-kernel use network devices to implement network protocols and communication

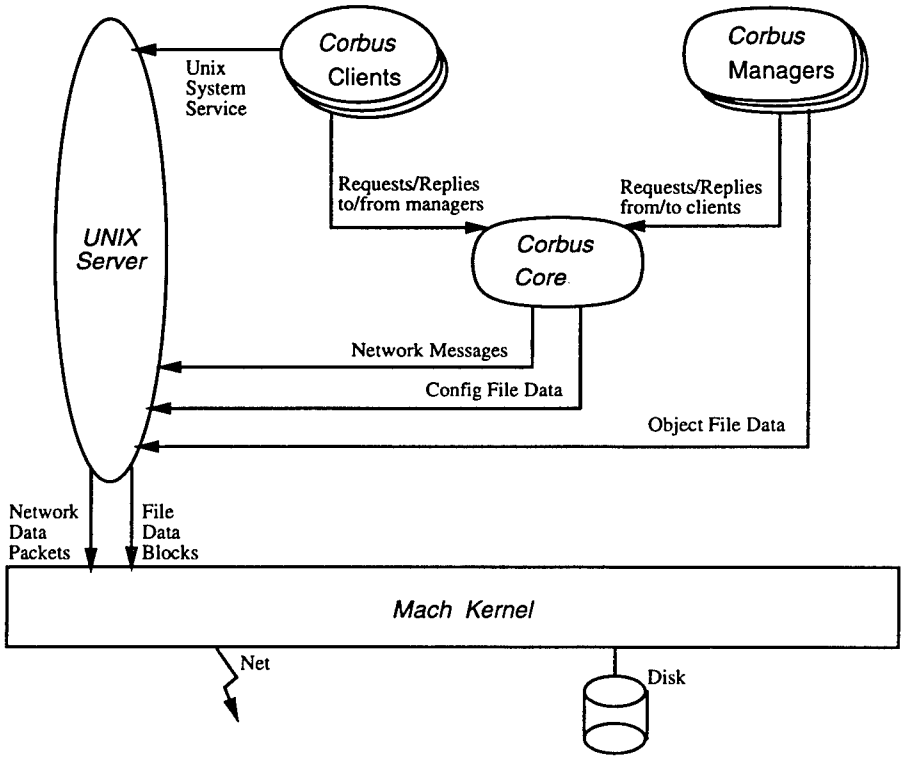

Figure 2.7: Dataflows of Corbus/Mach and Unix Server

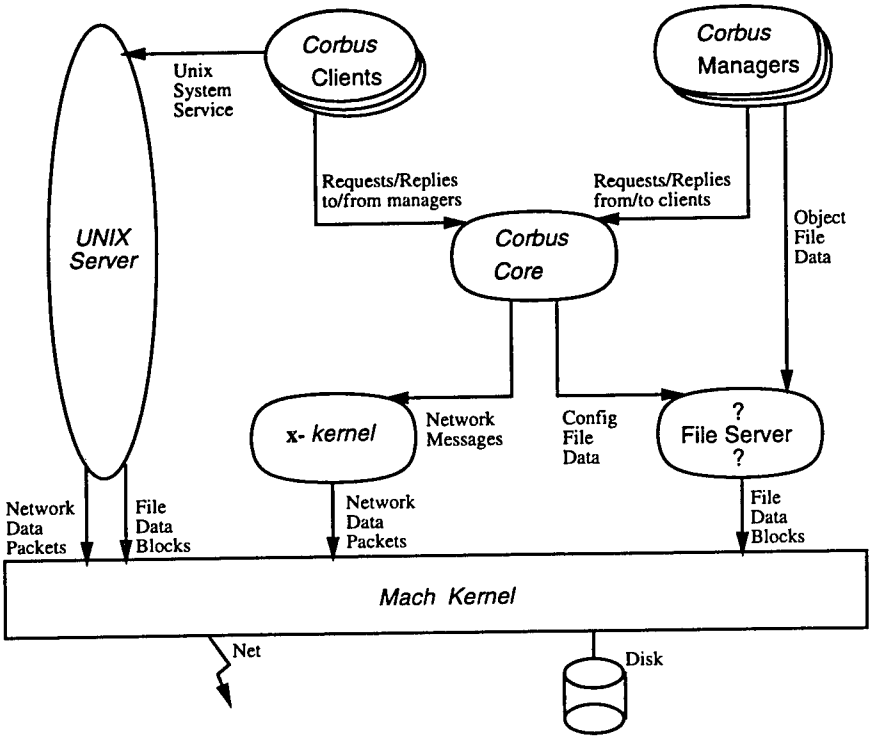

Figure 2.8: Dataflows of Corbus/Mach

services. Likewise, O/S personalities and file servers use disk devices to implement file services.

2.5.2 The x-kernel Protocol Server

Research at the University of Arizona has produced the x-kernel [1], a general, extensible framework for modular implementation of network protocols. The existing x-kernel implementation of the IP suite can be used to meet Corbus's networking requirements, rather than having Corbus/Mach rely on an O/S's networking.

An additional and promising feature of the x-kernel is that it has been used as the basis of an implementation of Mach IPC in a network environment [2]. By using this MachNetIPC implementation in Corbus, the same Mach IPC service can be used for both local Corbus communication and for Corbus communication between Corbus/Mach hosts. This can be a significant change when considered in conjunction with potential Corbus/Mach IPC usage. Unlike most kernels on which Corbus is based, Mach is specifically designed to enable point-to-point communication between any two processes running on a single host. Therefore, it is natural to optimize local Corbus communication to enable direct client/manager communication via Mach IPC, and to largely eliminate the intermediary role of the Core wherever possible. Use of networked Mach IPC, in turn, allows remote client/manager communication without using the Core in an intermediary role. Because Mach was designed for this sort of communication, it would not carry the potentially heavy resource usage penalties that would accrue from arbitrary client/manager communication in other systems, e.g., exhaustion of available network domain sockets in a Unix system.

However, the Corbus/Mach Core would still need to facilitate communication with non-Mach Corbus hosts, and would therefore require standard Internet protocols. The x-kernel provides these protocols, and the Corbus Core could use them for non-Mach Corbus communication.

2.5.3 Independent File Server

Use of an independent file server would allow the Core and managers to avoid functional dependence on an O/S personality for file service. If a Corbus/Mach implementation used one particular personality for file service, then the use of another personality would require modification to the Core and manager software. A independent file server for Mach would obviate

this difficulty.

There are several candidates for an independent file server for Corbus/Mach. Both Hurd and Mach/US (described below) have a separate file server which might be usable separate from a complete O/S implementation. The Mach/US file server, in particular, was designed to implement a general, "personality-neutral" file service that operates in a separate task from other above-microkernel system services. Trusted Information Systems' TMach system [4] also includes a separate "personality-neutral" file server that could be built for separate use.

Another approach to Corbus file service employs a file service that is obtained from an O/S personality. Dependence on on O/S personality would be prevented by the use of minimal, portable, O/S-independent file service for Corbus. Rather than using an O/S file service interface directly, Core and manager software would use this O/S-independent interface. The goal for this interface would be to abstract away from any O/S-specific implementation details, so that the associated implementation of this interface would be cleanly separated from other Corbus software. There would be multiple implementations for this interface, each in terms of a different underlying O/S file service. Each Corbus/Mach host would use an implementation which is based an O/S file service provide by an O/S personality on that host. As a result, Corbus/Mach Core and manager software would have a common file service interface on all types of hosts, and would not be dependent on any one O/S personality for file service.

Such a compromise approach is feasible because file access is directly involved in the main work of an ORB: providing a client/server communication infrastructure. Therefore, it is still possible to enhance that infrastructure on Mach hosts in order to take advantage of communication-oriented advanced system capabilities. This would occur even though stable storage is provided by an O/S personality component that is not part of the Corbus system and would not be enhanced as part of Corbus/Mach.

The drawback of this approach is that no file system-oriented advanced capabilities can be brought to bear to enhance the capabilities of Corbus managers. (With respect to the Core, file system enhancements would likely make little difference because the Core makes minimal use of files.) However, use of advanced file system technology may be problematic not only for O/S personality software, but also for the candidates for independent file server described above. Just as it is not feasible to make Corbus-specific extensions to the file system of an O/S personality, it may not be feasible to make such extensions to independent file server software. Nevertheless, it is much more

likely that independent file servers, as research vehicles, may incorporate advanced capabilities that would benefit Corbus managers; work with O/S personalities focuses more on performance than advanced capabilities.

Finally, a new stand-alone Mach-based file server for Corbus is another approach to the integration of advanced file service capabilities. This approach would be attractive if desirable capabilities were not present in any Mach-based O/S personalities or file servers, but could be integrated into Corbus/Mach file server. The requirements for such a server would be modest, because Corbus uses only basic file access and manipulation features: the Core stores some configuration and state data in a small, fixed set of files; and each Corbus manager stores its object data in a single file for each managed type. Thus, the requirements for a hierarchical file system are minimal, as are the requirements for file manipulation, which consist of reading, seeking, and writing.

2.5.4 O/S Personalities

There are several O/S personalities currently available on Mach, and commercial O/S vendors are developing more. The following list is a summary of currently available personality software for Mach. In addition, DOS and Macintosh O/S personalities were developed for Mach 2.5, an older version of Mach. Note also that the Mach Portable Operating Environment (POE) is not on the list. Prior to the inception of the Cronus/Mach project, POE was a identified as the candidate for a Mach system base. However, POE failed to progress to being a viable robust system base.

Mach/UX is a Mach-based Unix environment available from CMU as part of the standard CMU Mach 3.0 distribution. Unix functionality is implemented in a single server running as a task on the Mach kernel. The Unix server is largely composed of Unix kernel code (and hence requires licensing), repackaged to run in a Mach task rather than in supervisor mode.

OSF Mach is actually several Mach/Unix systems available from OSF, all of which include the same sort of single-server, repackaged Unix kernel code as Mach/UX. Unlike Mach/UX, OSF systems are still active research vehicles.

Mach/US is a Mach-based Unix environment recently made available from CMU [3]. A result of several graduate students' thesis work, Mach/US

is an almost complete reimplementation of Unix using multiple separate servers and an emulation library. Mach/US is a recent development and is not as stable as more established systems.

Hurd is the Free Software Foundation's (FSF's) Mach-based operating system that is available under the FSF's GNU General Public License [6]. The general motivation behind Hurd is to provide a Unix implementation, but one that is easily customizable both by reconfiguration of system components, and by user environment customization. Although Hurd has some distinctive features that would be useful for Corbus, experimentation with Hurd outside of FSF has shown that Hurd is not yet complete or robust enough to support non-trivial application software such as Corbus.

Lites is a Mach-based Unix single server that is composed entirely of unencumbered code, primarily from the 4.3BSD-Lite distribution [7]. repackaged for Mach by graduate students at the Helsinki University of Technology, the University of Utah, the University of California at Berkeley, and CMU. Although ongoing work addresses robustness of some less central areas of Unix functionality, Lites is complete, released, and in daily use by several Mach research groups.

Chapter 3

Mach IPC for Corbus Communication

In the Corbus/Mach architecture described in the previous section, the communication functionality of Corbus was exactly the same as in the standard Corbus architecture. The key features of the Corbus/Mach architecture are Mach-specific changes to the system components that Corbus components use for system service. Having made the first architectural step to use of Mach features, it is then possible to change the functionality of Corbus/Mach to make more sophisticated use of Mach features. This section describes the first of those changes: the use of Mach IPC for direct client/server communication. This change to the basic Corbus communication architecture enables other changes.

3.1 Background and Goals

The primary role of the Corbus Core is to facilitate communication between clients and servers of distributed object-oriented applications. Therefore, Mach-specific enhancements of Corbus communication capabilities would provide direct benefit to a primary area of Corbus functionality. However, it is essential that such enhancements do not affect the overall functionality of Corbus, either in the way that applications interact with Corbus, or in the way that Corbus meets its overall system goals, e.g., heterogeneity and scalability.

Therefore, Corbus/Mach communication enhancements must follow two goals:

- All functional changes must be "under the covers" of the application programming interface (API) implemented by Corbus, so that application software can work as is on Corbus/Mach systems and still reap the benefits of Corbus/Mach communication enhancements.
- All functional changes must preserve the use and correct operation of existing Corbus mechanisms for heterogeneous communication, replication, location, and so on.

To meet these goals, Mach IPC changes to the Corbus communication infrastructure will be limited to the interaction of the following three components of Corbus:

1. The Corbus Core, which runs as a separate process that acts as a message switch and object locator;
2. The Corbus Client Communication Library (CCL), which implements client object requests in terms of communication with the Core; and
3. The Corbus Server Communication Library (SCL), which implements the server side of client/server communication in terms of communication with the Core.

These three components remain in Corbus/Mach, but the interactions between them can be optimized significantly by using Mach mechanisms. But because the changes effect only the interactions but not the application API, the changes meet the first goal above. The second goal is met because these changes do not affect the purpose or mechanisms of other parts of Corbus, e.g., canonical data typing for heterogeneous communication.

3.2 Basic Approach

The basic change to Corbus communication on Mach is to allow Corbus clients and server to communicate without their messages being switched through the Core. Thus, the role of the Core would change from being a message switch (which includes object location functionality) to a location broker.

3.2.1 Message Switch vs. Location Broker

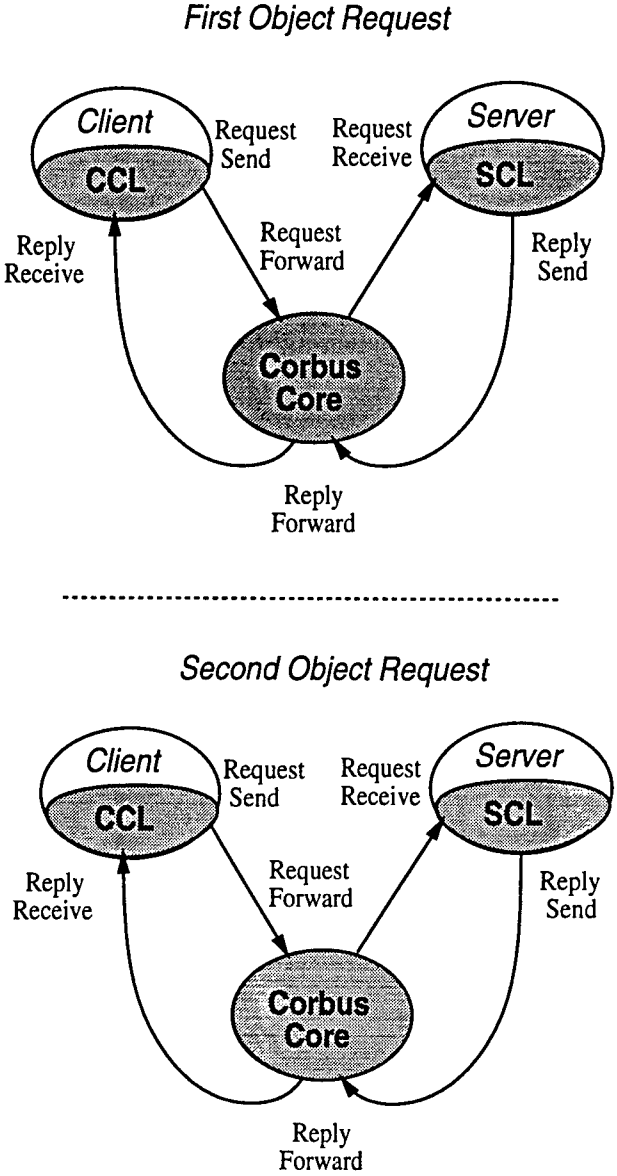

Figure 3.1: Standard Corbus Communication Architecture

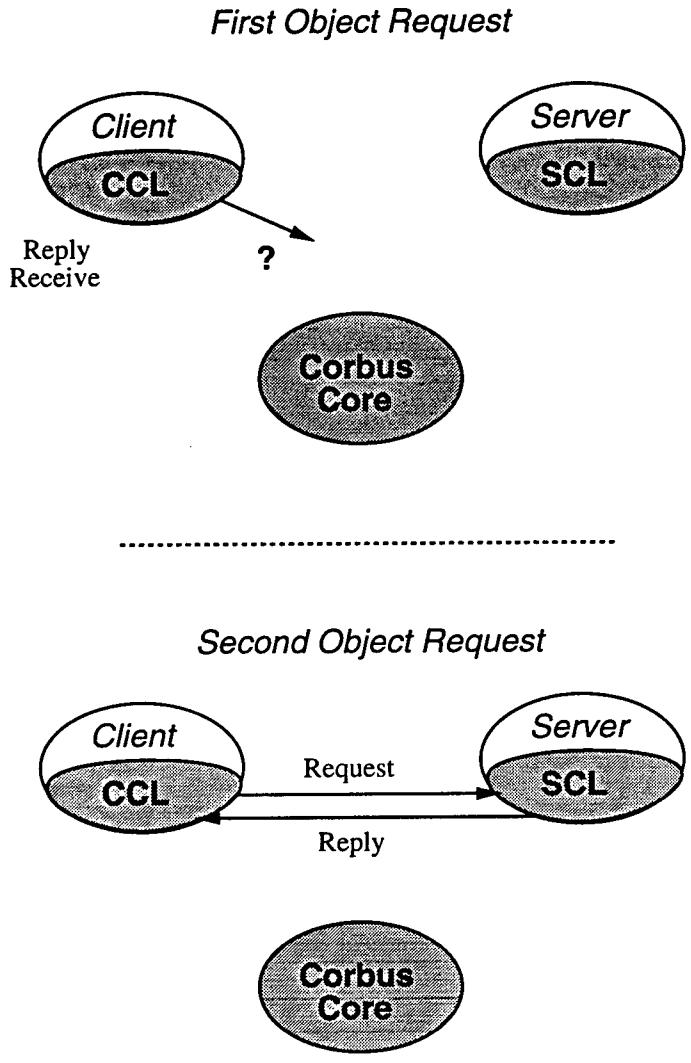

Figure 3.2: Corbus/Mach Communication Architecture

To understand the nature of the change in the Core's role from message switch to location broker, it is necessary to understand the distinction between the two roles. This distinction is illustrated in Figures 3.1 and 3.2. Both figures show the relationships of three processes: a client, a server, and the Core. Figure 3.1 shows the relationships in the standard Corbus communication architecture. When the client makes an object request, the CCL sends a request message to the Core, which receives it and forwards it to the server that manages the requested object. The server processes the requested operation on the object, and computes the result of the operation. Then SCL sends the result in message to the Core, which receives it and forwards it to the client that made the request. This series of steps happens for the first interaction between a given client and a given servers, and also for subsequent interactions. During those subsequent interactions, there may be less work for the Core to do in terms of object location and contacting the server; however, the dataflows between the three processes are the same.

Figure 3.2 shows the relationships that apply with direct client/server IPC. The right half of the figure shows that for subsequent interactions, the client and server communication without the intervention of the Core. The client's CCL constructs messages and sends them to the server, and the server's CSL receives them just as if they had come from the Core as in Figure 3.1. Figure 3.3 shows the significant savings in local IPC overhead that is realized with direct client/server communication— half as many kernel calls and context switches with direct client/server communication (on the right side of the figure) as with message switching (on the left side of the figure).

However, Figure 3.2 does not show how this direct client/server communication connection came into being. This communication initialization would have happened as part of the processing of the first message between the client and the server. The Core has an important role to play here, because the Core has the ability to locate the correct server for the client's object request, while the CCL does not. Therefore, there is some interaction between the CCL and the Core for object invocations for which the CCL does not know the appropriate server. The nature of that interaction is central to understanding the Corbus/Mach Core as a location broker. Details of this role are described in Section 3.2.3, which will fill in the gap in Figure 3.2.

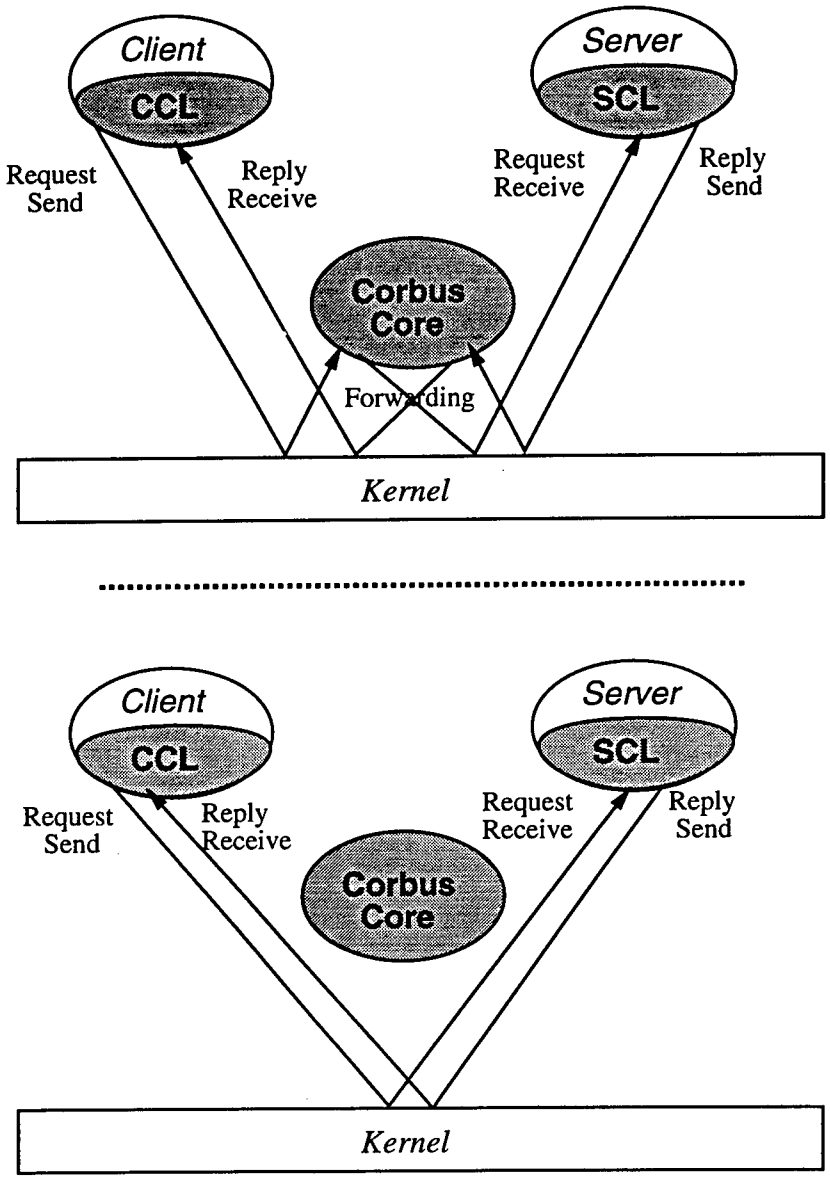

Figure 3.3: IPC Overhead of Corbus Communication

3.2.2 Core as Message Switch

Before discussing the advantages of the Corbus/Mach Core as a location broker , it is worth examining why the Core is a message switch in the standard Corbus architecture. The reasons are largely historical. Corbus is based on technology developed by a predecessor system, Cronus, which was developed for earlier O/Ss which were limited in two ways.

First, these O/Ss were not designed for general IPC between potentially large numbers of processes. For example, in early versions of Unix, the only alternative to parent/child process communication via pipes was via message queues or shared memory segment, of which there was a fixed, small number; as a result, there could be only a few pairs of processes in communication at any one time. Rather than having every client connect directly to servers, it made more sense to centralize control of scarce IPC resources in one message switch process with which every client and every server communicated.

Second, these O/Ss were also limited in terms of the amount of network communication that could be sustained. Again using Unix as an example, there was a small fixed number of sockets that could be open at any one time to provide a network connection between two processes on different hosts. With direct client/server network communication, there could be multiple clients connecting to a server on another host, thereby using multiple connections between the hosts. Again, centralized control of scarce communication resources made sense— one host's message switch could maintain a single connection with another host's message switch, and multiplex on that connection all client requests bound for the other host.

More modern O/Ss are less constrained in terms of communication resources. This is especially true of Mach, which was explicitly designed with IPC as a primitive feature on which most other system functions can be based. While network communication is not the province of the Mach microkernel, several network protocol servers have been build for Mach, of which the x-kernel is the most recent and successful. The x-kernel in particular is well-suited to avoid resources consumption problems because of its flexibility of configuration. For example, the x-kernel implementation of networked Mach IPC includes a protocol module which provides higher-layer protocols with connection semantics, while using caching techniques to maintain a suitable balance of actual host-to-host connections.[1]

[1]It is interesting to note that a similar connection-caching approach was adopted in Cronus, with the significant difference that instead of the system-wide optimization of connections of the x-kernel, Cronus only optimized usage of the subset of connections

However, Mach is hardly alone among O/Ss in being rich in local IPC and network communication resources. As is shown by the widely deployed OSF Distributed Computing Environment, it is now quite feasible to perform distributed computation in which clients connect directly servers. Therefore, there is a case to be made for a movement away from message switching that is independent of Mach. However, there are two features of Corbus/Mach that make location brokerage brokerage particularly attractive. First, with direct client/server communication being via Mach IPC, there other IPC-related features of Mach that can be brought to bear. Second, some of these features can be used to leverage off of the location caching approach of Corbus. Both these features are elaborated on in subsequent sections.

3.2.3 Core as Location Broker

The role of the Core as a location broker is to return to clients the information that they need to contact servers directly. Notionally, this would take the form of an object location request from the client to the Core; the Core would reply with the needed information, and the client would use it to send its object request message directly to the server.

However, there is some question as to the optimal implementation of location brokerage with the existing Corbus system. One way, of course, is to implement it as described above and shown in Figure 3.4. Another way is to add location brokerage to the existing message switching function of the Core. That is, the Core would switch messages when a client sends it a request, but in addition to the server's reply, the Core would also return location information. As a result, the CCL would send a request directly to a server if it knows how, but if not it would send the request to the Core, get the reply, and save the returned location information; the next time that client's CCL processed a request for the same server, it would use the saved location information to send the request message directly. This approach is illustrated in Figure 3.5.

There is little to recommend one of the approaches over the other, in that both have the same communication overhead. The first approach keeps the Core simpler, but then the Core already has the functionality needed for the second approach. Also, the first approach would require an addi-

available to the Cronus message switch. The lesson of this comparison is that the Mach/π-kernel environment provides more of this kind of transport-protocol-level functionality (and provides significant capabilities for adding more), leaving application-protocol-level software like Corbus free of these lower-level concerns.

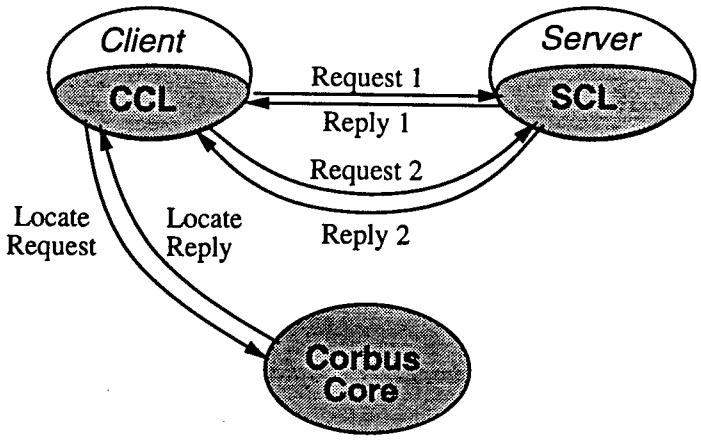

Figure 3.4: Core as Pure Location Broker

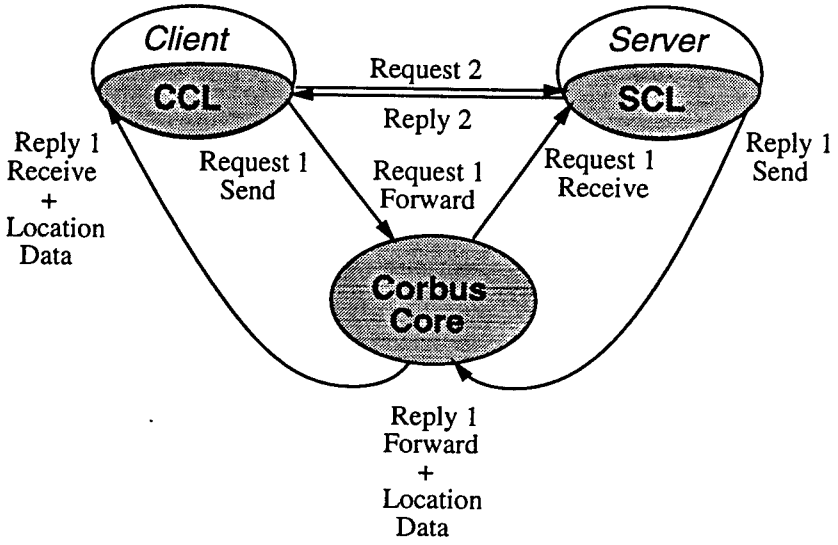

Figure 3.5: Core as Switch and Location Broker

tional "Location Request" message in the Corbus Operation Protocol (the protocol that the Core, clients, and servers all use to send object request and reply messages). Given that the switching functionality exists and the additional message type does not, it would seem preferable to adopt the second approach.

There is a third approach that would seem superior to both of these approaches. It eliminates one of the messages by having the server reply directly to the client. Object requests go from client to Core to server, but replies (with location information) go directly from server to client. This savings in overhead is potentially quite significant in that some clients may only contact servers a few times (or even only once) so that the cost of the first request is a significant proportion of the overall client/server communication overhead.

However, this third approach turns out not to be lower in overhead, because of relatively recent changes to Mach IPC mechanisms, described in the next section. Therefore, the third approach is not a superior alternative, and the second approach— retaining the Core's switching function for initial object requests which also serve as location requests— seems the best.

3.2.4 Synchronicity in Mach IPC and Object Requests

The third approach uses an asynchronous communication model, i.e., one in which messages are one-way communications between two points. This model matches that of the asynchronous IPC mechanism on Mach. However, Mach IPC is most often used for the synchronous communication between clients and servers, i.e., the client sends a request message and waits for a reply message. As with ORBs in general, Corbus client/server communication is usually synchronous, though with notable exceptions.

As a result, Mach IPC is in a somewhat unusual situation: asynchronous IPC is most often used to implement synchronous Remote Procedure Calls (RPCs) with paired asynchronous messages. Because of this situation, there has much work in optimizing Mach IPC for synchronous messaging, i.e., RPC. The result of this work is that most Mach systems under continuing development are now optimized for RPC. In terms of the third approach described above, three one-way messages turn out to have a higher communication overhead than two pairs of request/reply messages.

The optimization for RPC was originally developed in the University of Utah's thread migration work [9], which allowed the existing Mach message interface to be used for both asynchronous messages and synchronous RPCs.

In OSF's adaptation of the Utah work, the Mach message interface was retained as is for asynchronous messages, and a new kernel interface for RPCs was added. In commercial versions of Mach currently being developed (such as the IBM microkernel) the Mach message interface is retained, but used only for synchronous RPCs. The result is both a performance enhancement and a simplification of the Mach kernel. Asynchronous messaging is supported by building it from synchronous RPCs, in an inversion of the original Mach message approach. An asynchronous message is simply an RPC that deposits a message on the server side, and returns immediately to the client. Because of the use of thread migration techniques, this form of messaging is actually lower in overhead than the original Mach message mechanism.

In all these cases, both synchronous and asynchronous communication is supported. This is a critical result for Corbus. Although the majority of ordinary client/server communication is synchronous RPC, Corbus also uses asynchronous messaging. The Corbus asynchronous message facility, the "Futures" mechanism, is a key advancing computing techniques inherited from Cronus, and is also used to significant advantage in related systems such as Photon. Therefore, advances in Mach communication have enabled a significant performance benefit for the synchronous communication at the heart of Corbus ORB functionality, while still supporting the asynchronous client/server communication facility of Corbus. Further details of thread migration and Mach IPC will be discussed in Section 7.

3.3 Details of Approach

There are several significant details to the basic approach to location brokerage and direct client/server communication, particularly the details of usage of Mach IPC for interactions between the Core, the CCL, and the SCL. These interactions can be categorized into these areas: registration with the Core of both the CCL and SCL; use by the Core of registration information, for purposes of message switching; manager-to-client (SCL-CCL) communication; and client-to-manager (CCL-SCL) communication.

Prior to discussion of these four areas is a review Mach IPC concepts. Following discussion of these four areas is discussion of some additional refinements of the overall approach.

3.3.1 Mach IPC Concepts

Mach IPC is a mechanism whereby one task can send a message to another task over a communication channel called a port. Access to ports is granted and protected by the Mach kernel. A task can only use a port if the kernel has granted it a capability for that port. These capabilities, called port rights, come on two basic kinds: send and receive. A holder of a send right can send messages over the port, while a holder of a receive right can receive messages sent on the port by holders of send right. In addition to being used to send and receive messages, port rights themselves can also be sent in messages. When a port right is sent in a message, the sender loses the right, and receiver gains it. Such port right transfer is one of three ways that a port right can be granted; another occurs when a task creates a port and the kernel returns to the creator the receive right for the port; yet another occurs when a task creates a task, and the child task inherits the parent's port rights (unless the parent prevents this). In addition to being used and transferred, send rights may be duplicated, so that a holder of a send right to that port can transfer a send right for that port to another task, while still retaining a copy of the right. Receive rights may not be duplicated. As a consequence, potentially several tasks can send messages over ports, while only one task receives those messages.

Asynchronous message sends are commonly used to implement RPCs, in the following manner. The RPC caller creates a port (or reuses a port to which it already has the receive right) and uses it as a reply port. The RPC caller constructs the RPC call, bundles it into a message, attaches a send right to the reply port, and sends the message over another port. The RPC caller then waits on a message receive call using the receive right for the reply port. The RPC callee holds a receive right for RPC call port, and so receives the RPC message, including the reply port send right. After the RPC callee has done its RPC processing, it bundles the RPC results into a message, and sends it back to the RPC caller using the send right to the reply port.

Mach thread migration and true synchronous RPC mechanisms work in a similar but simpler manner, since no reply port is involved. A task that is willing to accept RPC calls uses a receive right to wait for RPCs, in a manner similar to the way in which a task uses a receive right to wait for an asynchronous message. A holder of a send right to that port uses the send right to send a message containing the RPC call. However, this send is also a blocking receive, all-in-one kernel call, unlike the two separate but

paired asynchronous unblocking kernel calls used to implement RPC. While the RPC caller is blocked, the callee receives the call message, performs the RPC processing, constructs a reply message, and then sends this message. The reply message send is not via a reply port, but via an RPC return mechanism that simulates the use of a reply port without the need for the client to explicitly create it, send it, and wait on it.

In subsequent detailed discussions, any synchronous communication is described without the details of the reply port, although either mechanism (true synchronous or paired asynchronous messages) could be used.

3.3.2 Corbus Registration

Every Corbus client and server must register with the Core, in order to announce itself as a user of Corbus facilities, and to enable the Core to deliver messages to it. The registration mechanism rests on the ability of any task to obtain a send right to a port received by the Core. Given that any task has such a port right, a potential Corbus client or server could use it to communicate with the Core and register. The first time a Corbus client or server performed used Corbus interface, the CCL or SCL would send a registry message to the Core and would include in the message a send right to a port received by the registering task. Additionally, servers would identify themselves as object managers and use type IDs to indicate the type(s) managed. Subsequent to such registration, the client or server could send messages the Core using the same port was used for registry. Servers receive object requests on the port supplied to the Core during registration. Clients receive replies on the port supplied to the Core during registration. If a Mach RPC mechanism is used, then clients do not require explicit port over which they receive replies.

Before leaving the topic of Corbus registration, it should be stated that it might be possible that every potential Corbus client or server could initially connect the Core. One common way to do this in Mach is via a local server registry service, with which the Core would register itself, and through which other tasks could lookup the Core and obtain a port to it. Of course, this local registry service would also depend on every task having a port to a local registry service task. This is usually accomplished by the mechanism of Mach system initialization and inheritance of port rights from parent task to child task. Mach system initialization is performed in part by the Mach init task, which is started by the Mach kernel at bootstrap time. All Mach tasks are descendants of the Mach init task and can inherit port rights from

it.[2] The init task can startup a server for the local registry service and can arrange for the inheritance of descendant tasks of a right to a port received by it.

3.3.3 Core Use of Registry Information

As a result of registration, the Core obtains and maintains two related forms of information about port rights. Each client and server has a Corbus ID (called a process UID). The Core associates a process UID with each port right it obtains from clients and servers during registration. The second form of port information concerns types. The Core associates each type with the process UID of the manager of the type. Therefore, the Core can map from to type to manager to port for communication with the manager.

This information is used by the Core when it switches messages. When the client makes an object request and the CCL hasn't yet contacted the manager of the object, the CCL sends the object request message to the Core. The Core attempts to determine the process UID of the manager of the object. This determination is made on the basis of a cache of mappings between object UIDs and manager process UIDs. If there is no mapping for the requested object, then the Core obtains the requisite information via the means of *object location.* In some cases, the result of object location will be a process UID of a local manager for which the Core already has a port. In other cases, the result of object location will be a process UID of a remote manager, and also a port right that can be used to contact the remote manager. In other cases, the result of object location will be a process UID of a manager on a non-Mach host for which there is no Mach port right.

In the first two cases, the Core uses the port right to send the object request message to the manager. Also, when the Core receives a reply from such a manager, the Core returns to the client not only the reply message but also the port right for the manager.

The reception of the reply from the manager is one area that is significantly impacted by the use of migrating threads for true RPC in Mach.

[2]The actions of the Mach init task are system-dependent, in the same manner in which the Unix /etc/rc.local file is way to describe system-specific bootstrap activities. In fact, on some Mach systems, the Mach init task simply starts a Unix server, and lets the rest of system initialization be done out the /etc/rc mechanism. Furthermore, the Lites system has a mechanism whereby a task can request a Mach port to the task running a Unix process, by specifying the Unix process ID. In any case, there are many possibilities for system initialization and registration.

In an asynchronous messaging-passing model, object request messages and manager reply messages are separate messages. For each kind of message, the Corbus Core has a separate interface to the CCL and SCL. For object request messages, the "Invoke" interface requires an object UID which is located as described above. The Core includes in its forwarded object request messages an additional datum: the process UID of the client that sent the message. For manager reply messages, the "Send" interface requires a process UID for the client to which the message is addressed. For correct delivery, manager's specify the process UID that was in the object request request message to which it is replying. When the Core receives a message via the "Send" interface, it does not locate the UID, but rather simply looks up the port associated with it, and uses the port to deliver the message.

In a synchronous RPC model, there is no need for an explicit reply port or an explicit interface for replies. Instead, message switching takes place in the form of two RPCs, one nested inside of the other. The first RPC is the CCL sending the object request message to the Core. While the client is blocking on this RPC, the Core (having located the object and obtained a port right to it) makes another RPC, this time to the objects manager. When the manager replies, it simply uses the kernel interface to return from an RPC; no explicit message send is required. As a result of the RPC return, the Core unblocks and receives the reply message. After adding to the message the manager's port, the Core likewise returns from the first RPC. As a result, the client unblocks, and the CCL receives both the reply message and the port right to use to contact the manager directly.

3.3.4 Server-to-Client Communication

The Corbus communication activity of servers consists of the CSL receiving receiving request messages, and sending replies to them. In the Corbus/Mach communications architecture, there are two cases of this activity. First, the request message is from the the Core, and the CSL sends the reply to it; second, the request message is from the client, and the CSL sends the reply to it. In either case, the CSL sends the reply to the same component that sent the object request being replied to. The issue for Corbus is whether the CSL needs to keep additional information in order to be able to reply to the same source as the request.

Figure 3.6 shows the role of the CSL in the standard Corbus architecture. All messages are received from and sent to the Core, so there is no need for the CSL to determine how to send the reply message. However, the server

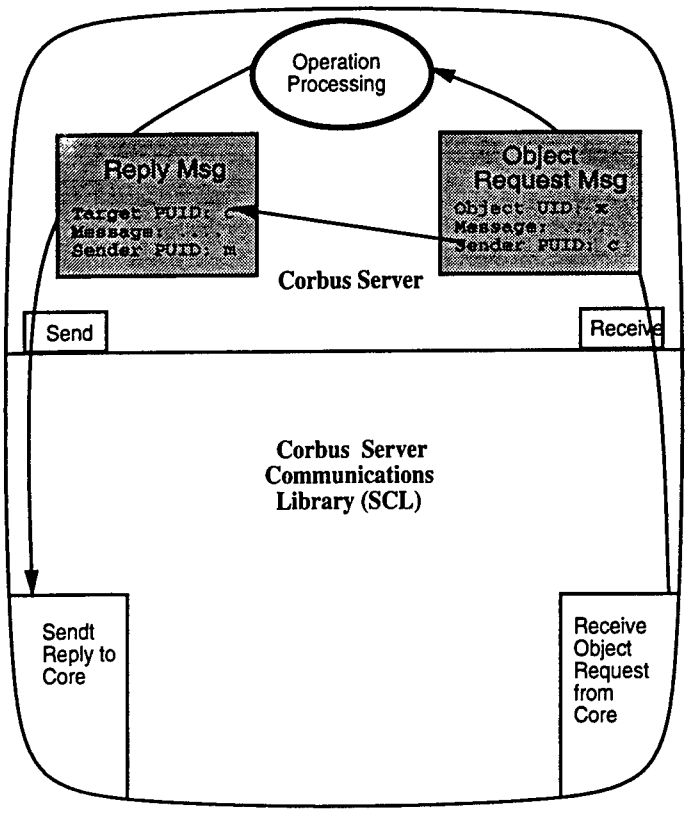

Figure 3.6: Standard Corbus CSL

software must tell the Core where to deliver the reply message. Therefore, the process UID from the request message's sender-UID field must be used as the target UID for the reply message.

Figure 3.7 shows the role of the CSL in the Corbus/Mach architecture, when asynchronous messaging is used to implement Corbus request/reply communication. Because the reply might go directly to a client, the request message might contain a reply port, of which the CSL must keep track. This is shown in Figure 3.7 as a table relating client process UIDs to reply ports from clients. When the CSL receives a Mach message from a client, it stores the enclosed reply port with the client's process UID in the table. When a reply is sent, the reply target process UID is looked up in this table; the resulting reply port is used to send the reply message. This extra work is necessary to keep track of separate destinations of reply messages; the multiplicity of destinations is the consequence of direct client/server communication.

Figure 3.8 shows the role of the CSL in the Corbus/Mach architecture, when synchronous messaging (RPC) is used. Because there is no need for a reply port, the server can send the reply message simply by returning from the Mach RPC. From there, the Mach kernel ensures that the reply message is delivered to the same task that sent the request message. This delivery is accomplished by means of a kernel implementation technique known as thread migration. When the client makes its RPC call and blocks, the client thread's execution resources are reused in the server; no scheduling context switch occurs. The kernel maintains, as part of the thread state, the client task to which the thread will unwind to when it returns from its processing in the server. Thus, so long as the thread that received the request message is also used to "send" the reply message via RPC return, the server does not need to keep track of clients. This result applies not only to client reply ports, but also to client process UIDs.

In order for Corbus servers to support the Futures mechanism, the Corbus server framework must support both forms of message reception, asynchronous and synchronous. The synchronous form uses RPC for efficient handling of the synchronous client/server interaction. The asynchronous form requires the use of reply ports so that the client can send the message (with reply port), continue processing, and eventually pick up the server's response using the reply port. On the server side, management of reply ports (as shown in Figure 3.8) is the only difference between synchronous object request RPCs and asynchronous Futures object requests.

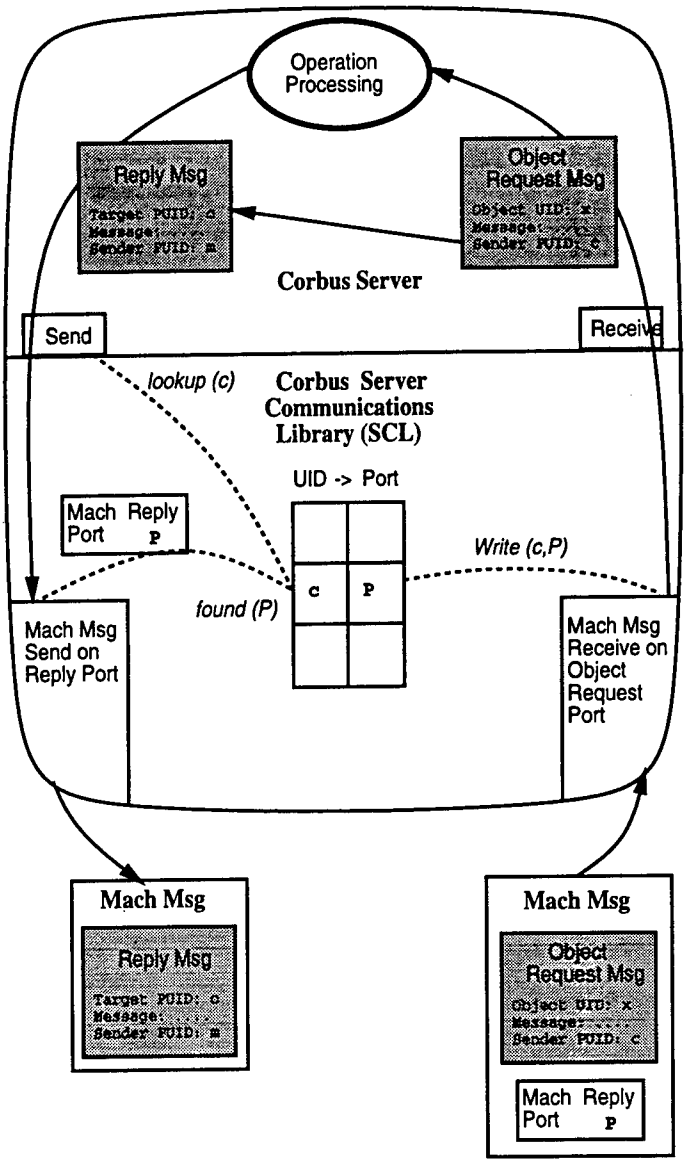

Figure 3.7: Corbus/Mach CSL with IPC

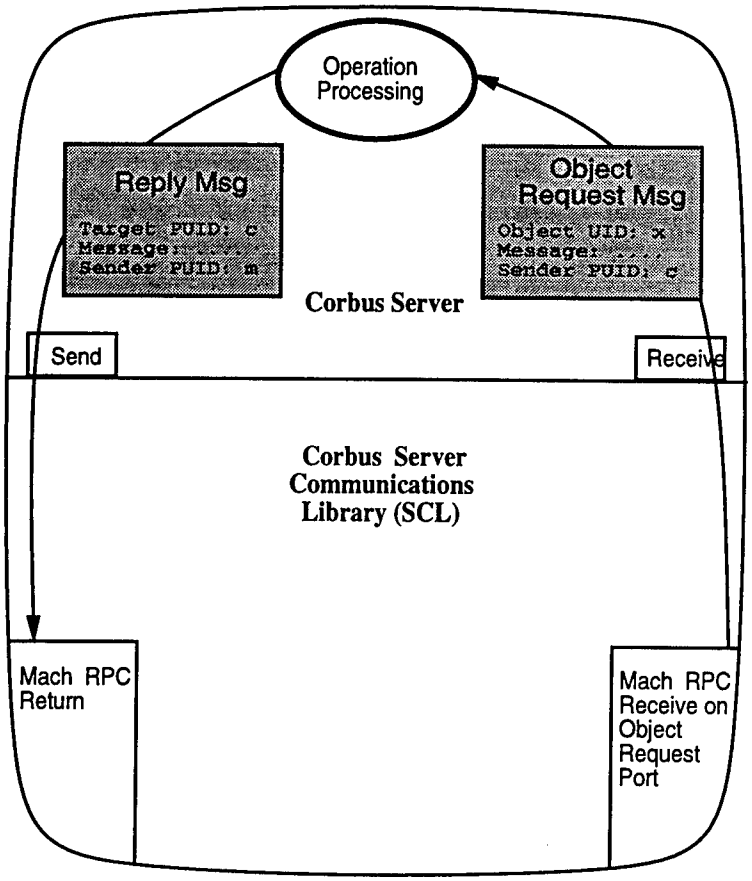

Figure 3.8: Corbus/Mach CSL with RPC

3.3.5 Client-to-Manager Communication

The Corbus communication from clients to servers consists of clients sending objects requests to object managers, and receiving manager replies. As described above, a client's CCL may or may not be able to directly contact a requested object's manager, and if not then the CCL uses the services of the Core to route the object request and receive location information. Therefore, one of the CCL's key functions in Corbus/Mach is handling object requests according to the following logic:

1. A client makes an object request using a standard Corbus application interface, and the client's CCL implements the interface with the following steps.
2. The CCL looks up the requested object's UID in a table of UIDs and port send rights.
3. If the UID is found, then the associated port is used to send the object request.
4. If the UID is not found, then the object request is sent using a send right to a port received by the Core.
5. When the reply arrives, the CCL checks the message for a manager request port send right. If found, then a new entry in the UID/port table is made. This new entry contains the UID of the object, and the port from the reply. Next time there is a request for this object, the port will be found and used as in step 3.
6. The CCL takes data from the reply message and returns to the client via the application interface.

Again, support for the Futures mechanism requires the use of asynchronous communication as well as synchronous communication. In the synchronous case of object request RPCs, the client API is simply a procedure call, inside of which a request is sent, the reply is awaited, and the reply returned. Such RPCs can be implemented either using true RPC with thread migration, or using pairs of one-way Mach messages.

In the case of asynchronous Futures calls, the client API is a procedure call to an invocation of a object request. Such procedures send off an object request message, and return a handle which can be used to claim the reply

at some later time. Such invoke calls are implemented with a message send, including a reply port. The returned handle is an API-level descriptor for the reply port. The claim of a Future is simply implemented as a message receive using the reply port.

Note that Futures functionality may also be implemented with migrating threads for RPC. An RPC could be used to convey the request message with reply port to the server, and immediately return to the client. Likewise, the server reply would use the reply port to use RPC to deliver the reply message to the client and return immediately to the server.

For the client to properly receive the reply message, the CCL would have to not only use a reply port, but also allocate kernel resources to "wait" to receive the reply RPC on the reply port. This "wait" is not done by the client thread that make the Futures invocation. The invoking thread simple performs a few kernel calls, and then proceeds with other processing. These kernel calls set up a kernel resource called an "empty thread" that enables an RPC to be received by using the replying server's thread resources. The replying server's thread executes in the client task, and delivers the message. The sequence of steps would be:

1. A client makes an object Futures invocation, and the client's CCL implements the interface with the following steps.
2. The CCL sends the request message, together with a send right to a reply port.
3. The CCL allocates kernel resources and binds them to the reply port so that the reply message can be received.
4. The CCL returns the Futures handle to the client.
5. The client continues its work.
6. Meanwhile, the server has completed the request, and used the reply port to send the reply message. The server thread migrates to the client, deposits the reply message, and returns.
7. Later on, the client software claims the future.
8. The CCL maps the futures handle to a port and a message received on that port. The CCL takes data from the reply message and returns to the client via Futures claim interface.

3.3.6 Objects, Managers, and Ports

In the foregoing discussion of CCL and Core interaction, the CCL uses the Core's object location service to contact an object's manager directly for any subsequent requests on the same object. While this is valuable optimization, it applies only to objects for which a client makes multiple requests. However, the optimization can be extended to cases where a client makes a request for an object that it has never requested before, but the object is managed by a server that the client has contacted before for service on another object.

This further optimization is based on the separation between the two tables described in Section 3.3.3: one table relating object UIDs and manager process UIDs, and another table relating manager process UIDs to port rights. The reason for the separation is that two objects can be managed by the same server, so the same port is used for the communication of requests on either object.

The same data separation of data can be performed in the CCL. Instead of keeping a single table relating object UID to port right, the CCL could keep two tables in the same manner as the Core. The Core's object location information, provided to the CCL, would include both the port right and the manager process UID. With this additional information, the CCL could perform a new function: if CCL were able to determine the process UID of the manager of a first-time-referenced object, it could check to see whether it already had a port right to communicate with the manager. If so, then the object request could be sent directly, even though it was the first time the client had requested that object.

Another way of viewing this potential for first-time object request optimization is that the use of Mach IPC allows the separation of two aspects of location brokerage: identification of the object's manager, and acquisition of communication resources to the manager. The distinction is an important one because the first aspect is purely informational in nature, while the second aspect requires a system call to acquire the communication resources. In Corbus/Mach, this acquisition would be means of IPC with the Core to obtain a send right to a port received by the manager. Likewise, a client might also use IPC with the Core to obtain object *location* data, i.e., the information about what server managed the requested object. If so, then there is no real optimization, since the Core must be consulted for every first time object invocation.

However, there is another way for clients to location information, using

a mechanism which, like IPC, is a core feature of the Mach kernel: shared memory. By sharing memory with the Core, clients can share the location information kept in cache by the Core. As a result, the CCL can do a cache lookup to determine whether it can find the object's manager and whether it already a port right for the manager. The CCL need only call on the Core if the object's manager hasn't yet been located or if the CCL doesn't yet have a port to the located manager. Details of this arrangement are described in the next section.

3.3.7 CCL Sharing of Location Cache

This section describes the use of Mach shared memory techniques for client access to cached location information. Use of cached location information by the CCL is based on the following observation: if a client's CCL already has a Mach port for a manager M as a result of having invoked on an object O1, and if the client can determine that an object O2 is managed by the same manager M, then the client can communicate directly with M for the *first* invocation on O2, rather than having to rely on the Core as a message switch.

There are a variety of Mach memory techniques that can be used for sharing of the location cache between the Core and a client's CCL. A critical factor of the sharing mechanism is the nature of the data access required. In the simplest access model, the CCL needs read-only access, while the Core needs read-write access, and writes by the Core must be readable by the CCL. While the out-of-line message approach is the simplest way to do quick sharing between two communicating Mach tasks, it does not ensure that ongoing writes are readable, unless the Core were to periodically refresh the CCL with a new message with out-of-line data for the location cache.

Rather than incurring the complexity of such refreshment functionality, a slightly more complex alternative involves the use of the location cache as a Mach memory object. Every piece of task virtual memory is part of memory object. Each memory object is managed by a memory manager, either the Mach default pager, or by an external memory manager such as a memory-mapped file server. Various tasks can map the same memory object, and shared access to a memory object yields common access to shared memory.

To set up shared memory-object access, the Core would arrange for its location cache to be in a region of its virtual address space inside one memory object, to which the Core has retained a memory-object descriptor port. When a client registers with the Core, the Core sends it a copy of the

send right to this port. Using this port, the CCL calls on the Mach kernel to map the memory object into the client address space. In order to minimize coordination between the Core and CCL, the memory-object is mapped in read-only mode. Subsequently, the CCL can read data from the location cache just the same as with any other region of the client's address space. Updates to the location cache by the Core will immediately be reflected in the client's address space.

Having set up this shared memory relationship during client registration, there is little or no overhead to either Core or CCL during subsequent operation. And because of the availability to the CCL of the location information, each client will have to use the Core as a message switch and locator significantly less. Core involvement will only be necessary in two circumstances: a client requests an object which is managed by a server that the client has not yet worked with a client requests an object that is not in the location cache.

Both these circumstances can be further optimized as described in the next sections.

3.3.8 CCL Location Caching

An additional refinement of the location mechanism is client-side caching of location information. That is, each CCL could keep its own private location cache. The reason for such private caches is that when there are many different clients doing requests many different objects, the amount of time an object's location stays in the Core's cache may be shorter than the time interval between repeated requests on that object by the same client. As a result, although the client is requesting the same object repeatedly, the object gets located repeatedly because its location doesn't stay in the shared cache long enough. This behavior can be particularly sub-optimal for clients operated by humans who examine several times each of a small number of objects

Therefore, a private cache would be way for client to save for itself the location information that it may need again later. As a result, the private cache becomes the primary repository of saved location information for a client; and the shared cache becomes a means by which one client can cheaply obtain location information previously obtained at some cost for another client. When an object isn't in the private cache, but is in the shared cache, the location information can simply be copied in memory from the shared cache to the private cache, thus increasing the likelihood that the

information will be available the next time the client needs it. In addition, when an object is not in either the local or the shared cache, the CCL sends the object request to the Core for object location and message switching. In such cases, the CCL can update its shared cache from the information in the reply.

With clients being able to rely on their own private cache, there would be less frequent turnover of the data in the shared cache, resulting in a greater likelihood that the shared cache information actually gets used by more than one client. This in turn maximizes the benefit of shared location information: because location information can be expensive to obtain, it is desirable make the information available to all clients once it has been obtained for one client.

It should be noted that although CCL private location caching involves extra processing and storage overhead, this overhead is not required. There could be different versions of the CCL, one that does no private location caching, and others that do caching, each with a different cache management approach. It would be an option open to client software developers to choose a CCL which performs private location cache management in the way that best meets the operational assumptions of the client software.

3.3.9 CCL Port Caching

Another refinement to CCL-Core location interaction is client-side caching of port rights to object managers. Without such caching, the CCL obtains manager port rights from the Core, one by one as the CCL contacts a manager that it hasn't contacted before. For clients which access a small number of managers a large number of times, this one-by-one approach is simple and adequate. For other clients, it would be advantageous to use client-specific information to anticipate which managers the client will need to communicate with, and to provide the CCL with the requisite port rights before object requests occur.

Such pre-supply of manager port rights could be done during client registration. The CCL could include in its registration request a list of types that the client will be using. (This list would be a build-time configuration item for the CCL, which the client software developer could choose to specify.) The Core would include in its reply a list of triplets: type, process UID of manager, and port.

The managers which the Core would include in such a reply would be dependent on system configuration and history. If the local host is running a

manager of a requested type, the Core will know about it as a result of manager registration. Object location of remotely managed objects is another way that the Core would know of a manager about which the Core could return information to a registering client. Another client may previously have requested an object of a type which a registering client has requested. As a result of locating this object, the Core will have obtained the process UID of and a port right for a manager of that type. If the object is still in the location cache, then the Core can include information about the manager (including the port right) in the registration reply. If the object has gone out of the location cache, however, then the Core would be unable to do so. To prevent such an occurrence, the Core could maintain a separate cache which contains information only about the manager, what type it manages, and a port for the manager. Since the number of managers is typically often than the number of objects, manager-only location information would tend to remain in its separate cache for a longer amount of time.

Chapter 4

IPC Distribution and Heterogeneity

The discussion in Section 3 does not completely address distributed application operation in an environment composed of heterogeneous hosts. This section addresses how the approach presented in Section 3 can be extended to such an environment.

The first significant issue is the use of the same IPC mechanisms between multiple Corbus/Mach hosts. Because the major goal of Corbus is support of distributed applications, i.e., client/server applications in which client and server may be located on different hosts, the advances described in Section 3 would be less significant if they applied only to Corbus functionality within one host. Fortunately, these advances can be extended into a distributed environment of multiple Corbus/Mach hosts.

The second and equally significant issue is that Corbus/Mach communication mechanisms must still support interaction with non-Mach systems that use Corbus. Such support, with some benefit from Mach-based technology, is also a critical area of Corbus/Mach operation in a distributed heterogeneous environment.

4.1 Distributed Mach IPC

The style of direct client/server communication described above can be extended to a distributed Corbus/Mach environment. The key technology for this extension is the implementation of a network communication service that provides the same features as Mach's local IPC, i.e., a transparent,

networked extension of Mach IPC. This *MachNetIPC* service is provided by a new component, the NetIPC Server. A NetIPC Server runs on every host to supplement the IPC functionality of the Mach kernel. As a result, there is a Mach IPC service that operates the same both locally and remotely; it is provided by the kernel and the NetIPC Server.

4.1.1 MachNetIPC and x-kernel

There have been several implementations of Mach IPC in a network, e.g., [13], but the most recent one, and the one that is the basis of continuing Mach work, is based on the University of Arizona's x-kernel [12]. The x-kernel operates as a protocol server for Mach, providing the network protocol service using the Mach kernel's device and IPC functionality. The MachNetIPC implementation is a set of Mach-specific protocol implementations within the overall protocol framework of the x-kernel. It is built on general-purpose protocols also provided within the framework. These general-purpose protocols are also directly available, most notably the Internet protocols used to communicate with non-Mach Corbus hosts. Thus, the x-kernel functions both as a MachNetIPC Server for Corbus/Mach and as a Protocol Server which enables non-Mach interaction within a heterogeneous environment. Just as significant, the x-kernel's dual role also allows it to be the focus of Corbus/Mach usage of other Mach-based advanced capabilities for distributed and networked systems.

The x-kernel's ability to be such a focus is derived from its basic purpose: The x-kernel is an implementation of a highly modular framework which can include an arbitrary layered graph of protocols. Among the protocols of the standard x-kernel distribution are common protocols, e.g., TCP/IP, but the purpose of the x-kernel is to provide easy extensibility by the addition of new protocols.

Flexibility is also a key factor; a protocol graph can be configured to use different protocols under different circumstances for optimal performance. For example, rather than a one-size-fits-all transport layer, a lightweight protocol can be used for communication between hosts on the same ethernet, while TCP can be used when the communicating hosts are internetworked.

Finally, the implementation of the x-kernel framework itself (rather than the protocols within) concentrates on addressing issues common to protocol layering—such as optimization of data buffering, copying, marshaling, etc.—so that protocol implementations need not do this. Thus, the flexibility, extensibility, and efficiency of the x-kernel make it an excellent vehicle for

developing and deploying protocols at all levels, as well as configuring them as needed for specific host configurations.

An additional point about the x-kernel is its nature as a server running on the Mach kernel. While this is very beneficial in the sense that the Mach kernel is not modified to add new functionality, there is the potential performance drawback that the network protocol software isn't running in the kernel's hardware state that permits direct access to the network device. However, just as there have been various approaches to NetIPC implementation, there have been several approaches to overcoming the lack of direct device access. There are a few such approaches that are currently being used, and could be used on a Corbus/Mach host. One of them, with which the x-kernel has been tested, is kernel co-location. In this approach, a task can run in the same address space as the kernel— without any modification to the code—by virtue of Mach kernel calls being short-circuited into procedure calls to the kernel's trap handler. This permits a significant efficiency gain without requiring a modification to the kernel [10].

A final point about x-kernel MachNetIPC is that it was specifically designed for transport of RPCs. The original implementation [12] was on Mach systems with asynchronous message IPC, but this implementation is the basis of work on Mach systems that include thread migration for true RPC. [11] [5] As a result of integration with thread migration, the x-kernel MachNetIPC will take advantage of a true RPC abstraction in Mach IPC and thereby implement a distributed RPC service for transport of RPCs between Mach hosts. Throughout this section, we refer to IPC and messages without distinction as to whether or not thread migration is used is used for RPC.

4.1.2 NetIPC Server

Clients and servers send and receive IPC using local IPC, regardless of whether the sender and receiver are on the same host. If they are on different hosts, then some network communication is necessary to move the data between the hosts. The NetIPC Server insulates message senders from knowledge about the locality of message destinations. Senders do not have to decide whether to use IPC for local communication, or network protocols for remote communication. Instead, both messages for both local and remote destinations are sent with the same interface. When communication is local, the sender holds a send right for a port for which the receiver holds the receive right. A message is sent via Mach IPC and the NetIPC Server is not

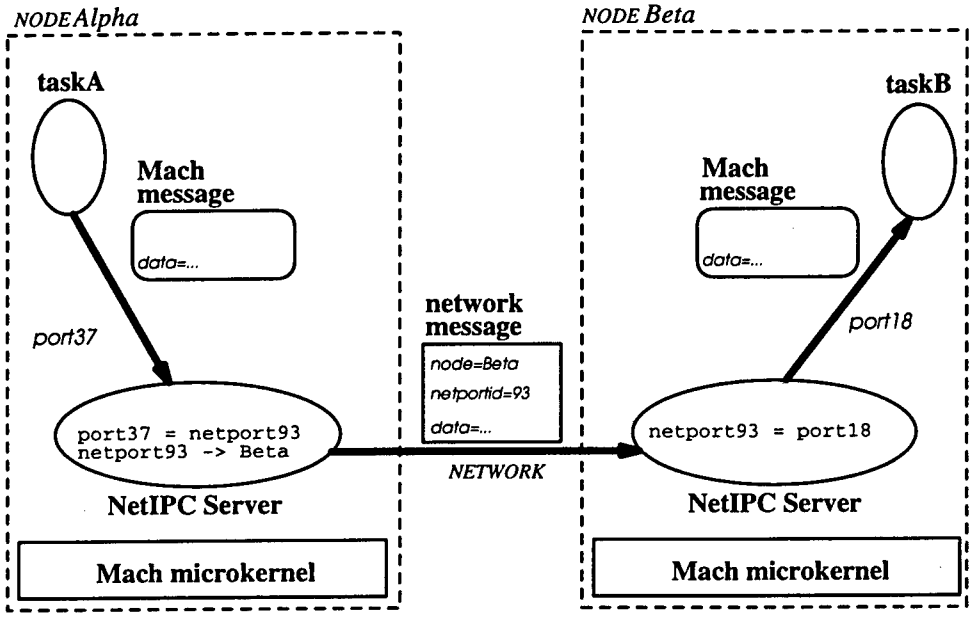

Figure 4.1: NetIPC Server

involved. When communication is remote, the NetIPC Server acts as a local stand-in for the remote task. On the sender's host, the NetIPC Server holds the receive right for the port that the sender uses— although the sender is not aware of the existence of the NetIPC Server. On the receiver's host, the NetIPC Server holds a send right for the port monitored by the receiver. The two NetIPC Servers use network protocols to move messages between hosts.

Figure 4.1 illustrates the basic role of the NetIPC Server for communication between Corbus-on-Mach hosts. A message sender sends a message and the local receiver is the NetIPC Server, which forwards the message to the NetIPC Server on the destination host. There, the NetIPC Server delivers the message to the receiver. As a result, tasks can simply use whatever port rights that have to send and receive message via local IPC; the NetIPC Server delivers remote messages, but does not interfere with local IPC.

Acting in this role of intermediary, the main function of the NetIPC Server is to maintain the state information needed to map local ports to global identifiers called NetPortIDs. The illustrated message transmission includes a mapping from the local port over which the message was sent, to a

NetPortID and the host which manages the NetPortID.[1] The NetIPC Server performs this mapping for each message it receives, appends the NetPortIDs to message (shown inside the network message box in the Figure 4.1) and sends the message over the network to the host that manages the NetPortID. There, the receiving NetIPC Server maps the NetPortID to a local port and delivers the message via local IPC on that port.

Because port rights can be transferred in Mach IPC, and because the IPC can span hosts, port rights must in some sense be transportable between hosts. In practice, this means that when a task does IPC to a remote task using a message including a port right, the NetIPC Server acquires the port right and the port bound to a NetPortID. Because port receive rights can be transferred in IPC, the destination host of a particular NetPortID can change. Thus, a significant part of the NetIPC Server's port management is tracking the movement of port rights.

A final point about NetIPC needs to be made about the mechanism by which a task can acquire a port right for a port that is managed by the NetIPC Server for the purpose of distributed IPC. Some sort of registry service is needed. In the x-kernel MachNetIPC implementation, a common Mach registry service is included, which uses network communication to allow lookups on other hosts. The simplest way to relate this service to how it would be used in Corbus/Mach.

Use of a registry service in Corbus/Mach is a four-step procedure. First, the Core on every host would register with local registry service and provide a port for the registry service to give to tasks doing a lookup. Second, the Core would look up other instances of the Core on other hosts. On each other Corbus/Mach host on which a host needs to communicate, the Core will perform a lookup on that Core host. Third, the local registry service will forward the lookup request to the requested host and that host's registry will return a send right for that host's Core. Because the registry service uses MachNetIPC for this forwarding and reply, the right associated with the port of the other host's Core (or rather a right for a NetPortID bound to that port) gets sent over the network in the lookup reply. The requesting Core receives a reply message containing a send right for a port held by the NetIPC Server and mapped to a NetPortID. Fourth, as a result of the registry and lookup, two or more host's Cores can communicate and freely transfer more port rights to one another. Therefore, each Core can transfer

[1]The NetIPC Server's maintenance of these mappings is shown as information about NetPortIDs and hosts inside the NetIPC Server bubble.

port rights that can be used for distributed IPC to other tasks on their hosts, e.g., Corbus clients and servers.[2]

4.1.3 Core Usage of MachNetIPC

The Core's primary use of the network is to send Locate requests and receive Locate replies, and send forward object requests and receive object replies. When a Corbus/Mach host's Core performs either location or forwarding to another Corbus/Mach host, the communication can take place via MachNetIPC rather than by the Internet protocols that standard Corbus uses.[3]

The benefit of MachNetIPC is that port rights can be transferred, to enable more direct communication. There is little extra protocol overhead needed to do this. In standard Corbus, there are two cases of object request transmission: local and distributed. In the local case, the request is send in the Corbus Operation Protocol (OP) via local IPC. In the distributed case, the OP message is embedded in a Corbus Inter-Host Protocol (IHP) message, which is sent via TCP/IP. In Corbus/Mach, the OP message is sent via Mach IPC in both cases. The difference is that in the distributed case, the message is forwarded to another host by the NetIPC Server, without the need for IHP. The MachNetIPC protocol is analogous to IHP, in the sense that both are protocols used to carry OP messages between hosts and that both use TCP/IP.

Figure 4.2 illustrates the primary ways that MachNetIPC is used by the Core. Distributed message switching is used as an example. The scenario involves two hosts, A and B, a client on one, a server on another, and a Corbus Core on both. There are four steps in dealing with an object request. In the first step, the client makes an object request and the CCL, not knowing the manager for the object, sends the request to the Core.

In the second step, the Core also does not know the location of the object and so, performs a Locate operation. Part of the Locate operation is the exchange of messages between the the Core of Host A (the client's host) and the Core of Host B (the server's host that manages the object).

[2]Note that this communication initialization is little different than a standard TCP communication setup via a well-known TCP port number. A bind to the well-known port number is like registering with the port number as that name, and the lookup on another host is like a connect to that host using the well-known port number. The difference is that the registry protocol is layered on top of MachNetIPC and thus the communication between hosts facilitates Mach IPC communication (including transfer of port rights) between the hosts.

[3]Internet protocols are used as transport protocols for the MachNetIPC protocol.

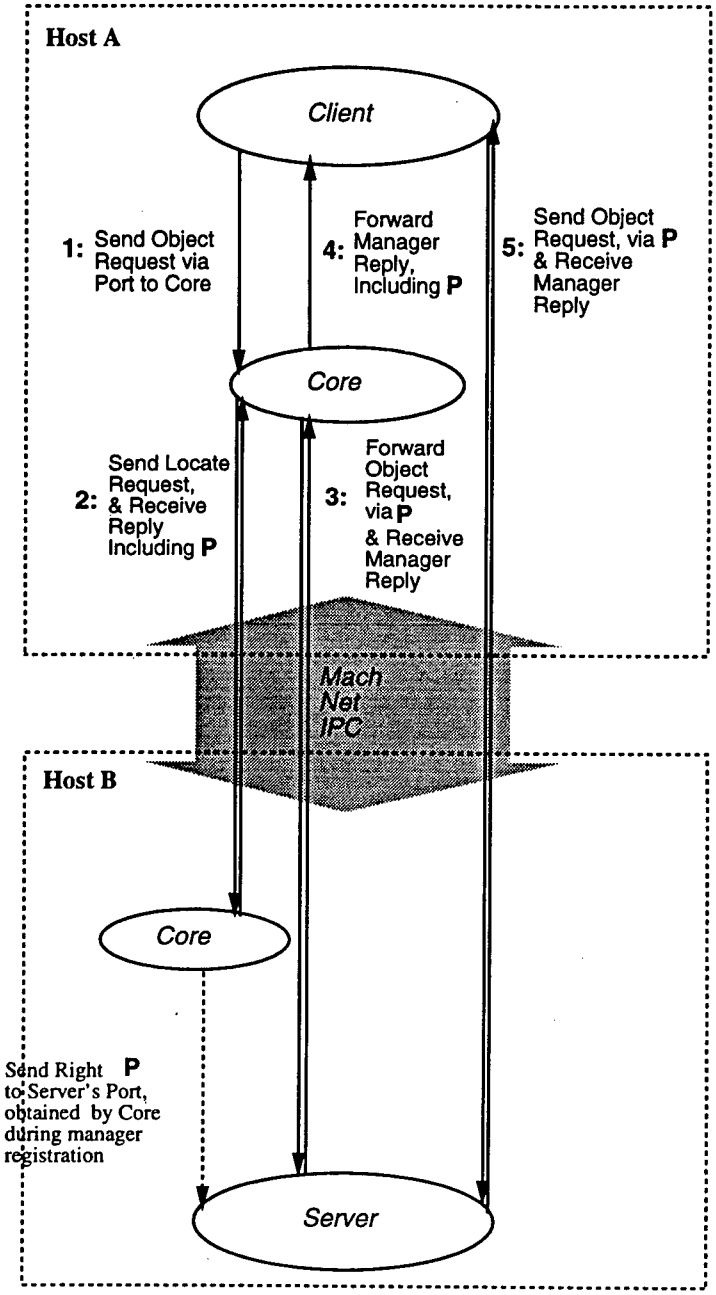

Figure 4.2: Distributed Object Location

The messages between the two Cores are transported via MachNetIPC. This communication uses ports that were set up during initial contact between the Cores of different hosts. In the reply to Host A's Core, Host B's Core includes a send right to a port (**P**) which is received by the server that manages the object being located. Host B's Core received a send right to this port when the server registered with the Core as the manager of the object type of the object being located. MachNetIPC translates this send right for **P** on Host B into a send right for a port **P** on Host A.

In the third step, Host A's Core uses the newly-acquired send right for **P** to forward the client's object request to the server and to receive the reply. Again, MachNetIPC is used to transport the message from the Core on Host A to the server on Host B and vice versa.

In the fourth step, Host A's Core forwards the reply to client, thus completing the object request/reply. The Core adds a send right for **P** to this forwarded reply message. As a result of the client's acquisition of this port right, the client can later use MachNetIPC to contact the remote server without the intervention of either host's Core. Such an object/request reply is shown as step 5 in Figure 4.2.

As a result of using MachNetIPC to exchange port rights, two processing steps have shorter dataflow paths. The path of step 5 is shorter than the standard Corbus path for client to remote server.[4] This the distributed analog of local direct client/server IPC without the assistance of the Core. The path of Step 3 is shorter than the standard Corbus path for Core to remote server.[5] In both of these comparisons, x-kernel is used in the same role in Corbus/Mach that an O/S kernel is in standard Corbus. This is because the x-kernel and an O/S kernel are software that implement network protocols and use a network device for communication. Also, both can run in a privileged processor state by means of the Mach kernel/server co-location mechanism mentioned earlier.

4.2 Heterogeneity in Corbus/Mach

Corbus/Mach hosts can operate in a heterogeneous environment, including hosts of various hardware architectures and O/S platforms. This heterogene-

[4]The standard path is client to Core to kernel/network/kernel, then to Core and to server. The Corbus/Mach path is client to x-kernel/network/x-kernel, then to server.

[5]The standard path is Core to kernel/network/kernel, then to Core and to server. The Corbus/Mach path is Core to x-kernel/network/x-kernel, then to server.

ity has two effects on the use of Mach IPC mechanisms in Corbus. First, the use of Mach IPC mechanisms must account for differences in hardware architecture between Corbus/Mach hosts. Second, the use of Mach IPC mechanisms must encompass the use of the existing communication mechanisms on non-Mach hosts and Corbus software on those hosts.

4.2.1 Heterogeneity of Corbus/Mach Hosts

A Corbus/Mach distributed computing environment should be able to include Mach hosts of various hardware architectures. Multi-architecture support for client/server distributed communication is already a part of Corbus. The Corbus *cantypes* facility allows Corbus messages to be sent between hosts in such a way that the data format of the messages is interpretable by hosts of any hardware architecture. All messages are converted from the native data representation of the sender's host to the canonical hardware independent representation of cantypes. Likewise, received messages are converted from cantype format to native format before being processed.

However, the use of cantypes sometimes imposes unnecessary overhead. When a client and server on the same host exchange messages, cantypes are not necessary because both client and server have the same data representation. The same is true when the client and server are on different hosts, but both have the same hardware architecture.

This extra overhead can be eliminated in many cases by using extra information about the destination of messages. This technique is not particularly dependent on Mach facilities and can be adopted by Corbus on other systems. However, the use of Mach IPC in Corbus/Mach allows this technique to be easily implemented.

The simplest case of heterogeneity to address is when both the client and server are on the same host. When a client's CCL sends a message to a server that it knows is local, it can dispense with cantype conversion. Instead of cantype conversion, the CCL sends the message data in local data format shared by the client and server stubs. Additionally, the CCL must include information in the message that indicates that it is not in cantype format. Based on this indication that the request is not on cantype format, the server's SCL would not convert the reply into cantype format.

The main issue with this approach is the means by which the CCL knows that cantype conversion is unnecessary. The Core can convey this information to CCL, in those cases where the CCL uses the Core for message switching and object location. The Core knows when an object's manager

is local, and in such cases, the Core can indicate this as part of the location data that it returns to the CCL. The Core knows when an object's manager is local because of two facts: (1) the Core knows about all local managers via Corbus registration; and (2) the Core performs object location. When it finds a server that manages an object, it can tell when the server is local because it knows all the local managers.

The next case of heterogeneity to address is when client and server are different hosts of the same architecture type. In such cases, it is necessary to identify the architecture type of the host for each Corbus/Mach manager. One way to make this identification is for the CCL, SCL, and Core to add an architecture tag to each message.[6] However, it would suffice for such a tag to be included only in locate reply messages from the Corbus/Mach Core of one host to another. The Core receiving such a message would not only keep the port right associated with the manager to which the location reply message refers, but would also keep the architecture tag. Then, when the Core forwards an object reply to the CCL, it can determine if the object manager is running on a host with the same architecture type as the local host. This determination would be based on a comparison of the tag value with the local architecture type. When the tag matches, the Core would indicate the to CCL that cantype conversion is not needed. This indication would be added to the object reply message along with the port right for the client to use to contact the server directly. As a result, the CCL would forego cantype conversion.

A final approach to architecture typing is to forego the use of architecture tags in favor of out-of-band information. That is, each Core would have as part of its configuration, a list of host addresses for hosts that have the same architecture type as the local host. Then, instead of checking an architecture tag in object reply messages, the Core checks the source host address and compares it to the list.

An additional note about this approach to cantype optimization concerns interaction with non-Mach Corbus hosts. When the Corbus/Mach Core interacts with the Corbus Core of a non-Mach host, the non-Mach Corbus may not have CCLs and SCLs that can accept non-cantype encoded messages. Hence, the Corbus/Mach Core must take care to inform the CCL about an like-architecture server only when it knows that the version of Corbus on the other host is a version that supports this optimization. Therefore, it is important for the Core to check software version number as well as architec-

[6]The use of such a tag in MachNetIPC is discussed in [12].

ture type. As with architecture type, the version information can be either sent in the message itself or stored out-of-band in each Core's configuration data.

4.2.2 Distributed Communication With Non-Mach Corbus

Corbus runs on a variety of O/S platforms; this variety is another form of heterogeneity that Corbus/Mach must accommodate. The Corbus/Mach Core must distinguish between Mach and non-Mach hosts and interact in an appropriate manner with hosts of each kind. There are two main features of Corbus/Mach communication with Corbus on non-Mach systems. The first feature is the use of the standard Corbus communication architecture, in which the Core is a message switch for all object requests and replies. The second is the use of the standard communication protocols used by non-Mach Corbus Cores. The distinction between Mach and non-Mach hosts occurs in four primary contexts:

Core Communication Initialization: When the Corbus/Mach Core initiates communication with another host's Core, the Corbus/Mach Core must first check whether the other host is a Mach host. If so, the communication can be initialized via MachNetIPC, as described in Section 4.1.2. If not, then communication must be established using TCP.

Object Location: When a Corbus/Mach Core sends out an object location request to another host, it must do so using the form of communication set up during Core communication initialization and accept replies using the same mechanism. Also, the Corbus/Mach Core must be able to accept object location requests using both forms of communication. In replying to such requests from other Corbus/Mach hosts, the Corbus/Mach Core should include a manager port right. But in replying to non-Mach hosts, no such Mach-specific information should be added.

Object Request Forwarding: When a Corbus/Mach Core receives a location reply from a non-Mach host, the Core cannot add a port right for the manager to the forwarded reply. Therefore, the Corbus/Mach Core/CCL interface should allow this. In such cases, it will not be possible for the CCL to directly send a subsequent object request to the manager. This is because of the lack a port right. As in other

cases where the CCL lacks a port right for a manager, the CCL sends the object request message to the Core for forwarding. When the Core is forwarding an object request to a non-Mach host, the Core should use the standard Corbus communication mechanism established during Core communication initialization.

Object Requests Forwarded: In addition to sending forwarded requests to Mach and non-Mach hosts, the Corbus/Mach Core must be able to receive forwarded object requests from both host types, using either MachNetIPC or Internet protocols, as appropriate.

Because the Corbus/Mach Core must communicate with the Cores of both Mach and non-Mach hosts, the Core's state information must reflect the difference in host type. Actually, the difference in state information is rather slight. In the case of a non-Mach host, the Core must maintain the host's IP address and the TCP/UDP port number used to connect to that host's Core (usually the same established port number). In the case of a Mach host, the Core must also maintain the host's IP address in order to use it in a request to obtain a Mach port right to the Core on that host. Also, a port number must be maintained, although instead of a TCP/UDP port number, the number is the numeric handle for a Mach port send right.

A final issue for heterogeneous inter-host Corbus communication is the standard Corbus feature of direct client/server large message delivery. The feature is available for clients and servers to bypass the message switching function of the Core, though the Core (or Cores, if the client and server are on different hosts) must be involved in setting up the direct connection. Unlike the direct client/server connection of Corbus/Mach, this feature is suitable only for cases in which the volume of client/server traffic is large. This feature does not scale well, in the sense that it is not feasible for every client and server to have a direct connection. This is due to because of the expense that would be incurred by significant usage of communication resources. In Corbus/Mach, by contrast, direct client/server communication is tied to local IPC, and the x-kernel MachNetIPC implementation takes care of optimizing on such resource usage issues as the number of open TCP connections.

Nevertheless, direct large message communication is a part of the standard Corbus communication capability and Corbus/Mach must support it. Therefore, the Corbus/Mach Core must support the inter-Core communication necessary to set up direct client/server connections. Likewise, the Corbus/Mach CCL and SCL must support the use of TCP to accept and

initiate large message connections. As a result, a server on a Corbus/Mach host can accommodate a non-Mach Corbus client's request for a direct connection for large message delivery. As with the Core, the CCL and SCL obtain TCP service from the x-kernel.

4.3 Heterogeneity and Dataflows

This section summarizes the dataflows and architecture of a Corbus/Mach host, taking into account the effects of distributed Corbus communication via MachNetIPC and of direct large message delivery.

Corbus/Mach local dataflows are illustrated in Figure 4.3 which is a modified version of Figure 2.8. Figure 4.3 adds new dataflows (solid arrows) that are the result of the IPC optimizations described in this section. Also shown are the dataflows (dotted lines) that are retained from Figure 2.8.

There are four new dataflows. First, Corbus clients and object managers on the same host can use Mach IPC to communicate directly. Illustrated as the "Local object requests/replies" dataflow, these interactions are based on information from the Core which acts as a location broker. Second, Corbus clients can use the x-kernel's MachNetIPC service to communicate directly with object managers on separate hosts, as illustrated in the "Requests/Replies to/from remote Corbus/Mach managers" dataflow. Third, Corbus object managers can use MachNetIPC to communicate directly with clients on separate hosts, as illustrated in the "Requests/Replies from/to remote Corbus/Mach clients" dataflow. These interactions obviate the need for the Core to perform message switching for much of the distributed client/server communication between Corbus/Mach hosts.

Finally, the Core/client and Core/manager dataflows remain, but exist for the Core to switch object requests and replies for which the client does not have the ability to communicate directly with the manager. There are two distinct kinds of such cases. One case is when the object request goes to a manager on a Corbus/Mach host and the reply contains a Mach port right that enables direct client/server communication in the future. The other case is when the manager is on a non-Mach Corbus host and the Core must perform message switching for all object request messages.

Figure 4.4 shows a revised architecture with additional component interfaces that reflect these dataflows. Clients and managers now communicate directly with one another via local IPC. Clients and managers also use local IPC to communicate with the x-kernel in order to use the MachNetIPC

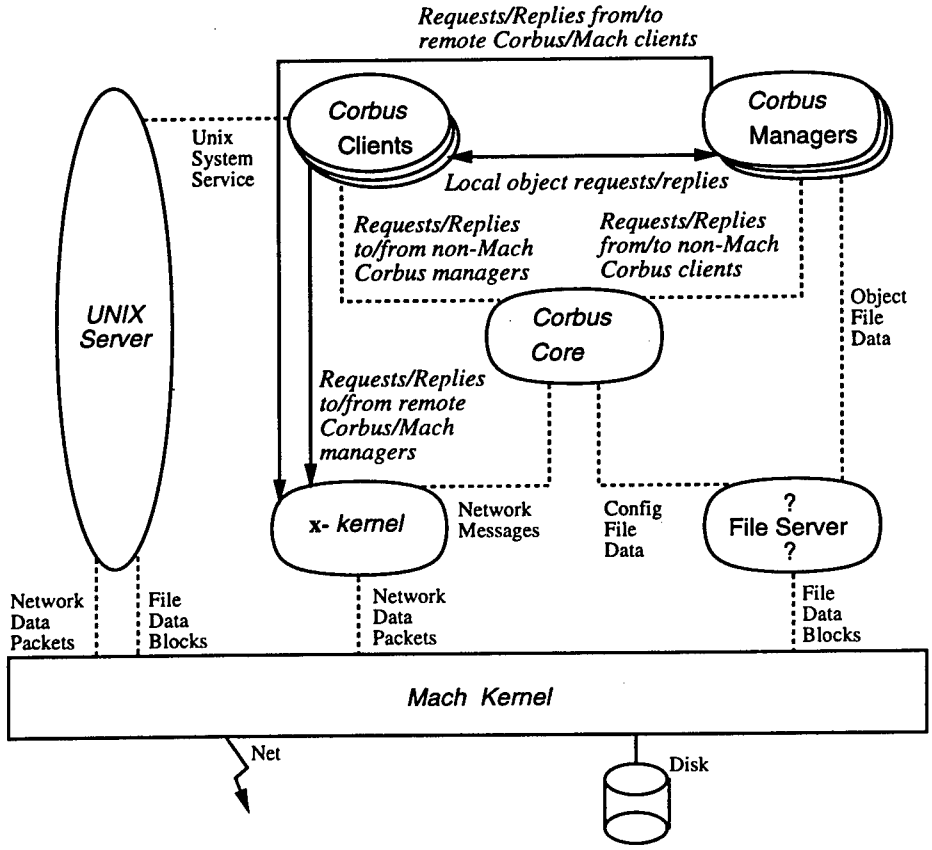

Figure 4.3: Dataflows of Enhanced Corbus/Mach

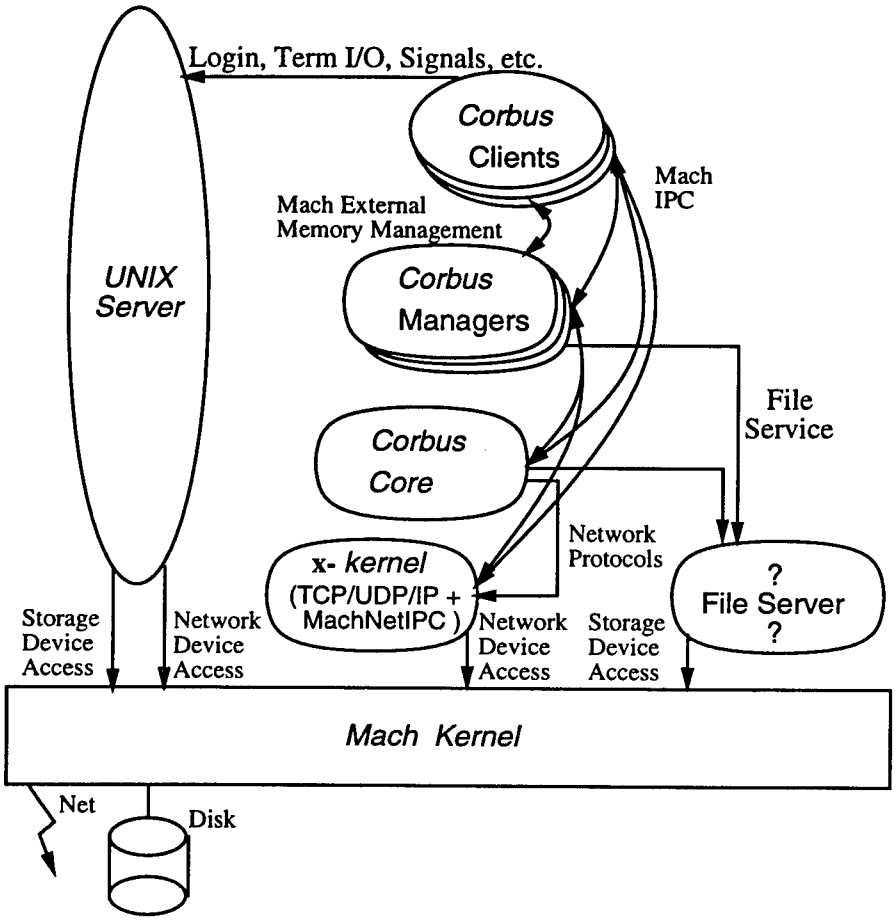

Figure 4.4: Enhanced Corbus/Mach Architecture

service to communicate with managers and clients on other hosts. Clients and managers still communicate with the Core for both object location and message switching to non-Mach Corbus hosts. Finally, there is a new arc between clients and mangers, labeled "Mach external memory management" to show a memory sharing relationship between clients and object managers. This new relationship is the topic of the next section.

Chapter 5

Mach Shared Memory for Corbus Communication

Previous sections of this report have focused on client/server interaction using a message-based paradigm in which communication takes the form of an RPC,[1] i.e., synchronous messaging between a caller/requester/client and callee/responder/server. This paradigm is generally useful because some form of local or distributed messaging is available or can be constructed in the environments that are used for client/server computing. However, there is another paradigm for client/server communication that is less general than messaging, but which is more powerful and beneficial for certain types of applications. This paradigm is based on memory sharing between client and server. A shared-memory approach is strongly supported by Mach's memory management features.

The following paragraphs describe a shared-memory approach for Corbus. The fundamental concept of operation is to use Mach memory management to allow a Corbus object manager to share memory with its clients. This sharing makes the data representation of objects managed by the manager available to clients. As a result, client software can access the object data directly. Such direct object data access contrasts with the indirect access of the message-passing paradigm, in which the object manager accesses the data on the client's behalf, as specified by the client in an object request message.

[1]The term Remote Procedure Call (RPC) is most often used loosely, without regard for whether a particular RPC involves one party of the RPC actually being remote from a another, in the sense of being on a different host.

5.1 Comparison of Shared Memory and Message Passing

A critical feature of shared-memory approach in Corbus is that it *does not change* the existing object request interface currently used by Corbus clients and servers. The same object request interface is used by a client, regardless of whether the client/server communication mechanism operates via message passing or by shared memory. In both cases, the interface is implemented by an autogenerated library specific to the particular object-oriented application of which the client and server are a part. Each such library implements the object request interface for the particular application, using the services of an underlying library for client communications. In message-passing cases, the application-specific library (often called a "client stub library") implements each kind of application-specific object request in terms of a message-passing communication library such as the Corbus CCL. In memory-sharing cases, the application-specific library implements the same request interface, but implements requests by direct access to and computation on shared memory containing object data.

An important distinction between these two styles of operation is the placement of the operation code. In a message-passing server, the server first receives an object request message from the client, and then performs the requested operation using the message data and the object's representation data. The result of the server's computation is sent back to the client library, which extracts result data from the reply message and returns the results via the object request interface. With a shared memory manager, the client has direct access to the object representation data; the client performs the operation using both the object data and the parameters of the object request. The results of this client-side computation are passed back via the object request interface, just as if the results had arrived in a reply message from the object's manager.

Because of these similarities, the client programming interface is unchanged by the shared memory approach, but library code is more complex. To avoid terminological confusion, we refer to this application-specific client library software by using the terms "client proxy code" or the "client's proxy." These terms are in distinction to the client application code, which is the new software that is written by the client developer, and which uses the object request interface provided by that application's client proxy. Thus, the term "client" refers to a process that is executing a body of software

composed of both client application code and client proxy code.

5.2 Client/Server Shared Memory Example

The following figures illustrate the two cases, and the various specific steps taken in each. Both show an example of client/server interaction in a fanciful application which implements operations on colored objects. In the example, a client invokes the operation GetColor on a specific object, referred to as Foo. There are no input parameters for the operation (other than the target object), and only one output parameter, which is stored in a data structure referred to as FooColor.

In both figures, the upper oval represents the client process, and the lower oval represents the manager process. Within the client process, a horizontal line separates the client application code (in the upper part) from the proxy library (in the lower part) that implements the interface to the manager's services. Within the manager process, a horizontal line separates the Object Database (ODB) code and data (in the lower part) from the rest of the manager (in the upper part) which implements the various operations on the object. The ODB implements access to the actual representation data of each object, and stores this data in files. Object data can be copied from storage to memory, and the memory used to perform operations on the data.

Figure 5.1 shows the usual Corbus message-passing approach. The client application code calls the GetColor procedure to request the GetColor operation on the object Foo, and to store the result in FooColor. (This call is illustrated as an arrow crossing the boundary into the proxy library, and returning back across the boundary to point to FooColor.) The GetColor procedure is implemented in proxy code which constructs and sends an object request message. (The message send is illustrated by the arrow from the client to the manager, the message in a box by the arrow.) The manager receives the message, and processes it. Part of the processing is to retrieve the representation data of the object that was invoked. The data is managed by the ODB, which stores object data in files. (The retrieval is illustrated as dashed lines which cross the ODB code boundary, and which connect the ODB data to the actual memory that contain the data after the retrieval has been done.) Once the data is retrieved, the operation is performed by calling the operation subroutine called DoGetColor. This DoGetColor subroutine is the code that actually performs the computation of the GetColor operation.

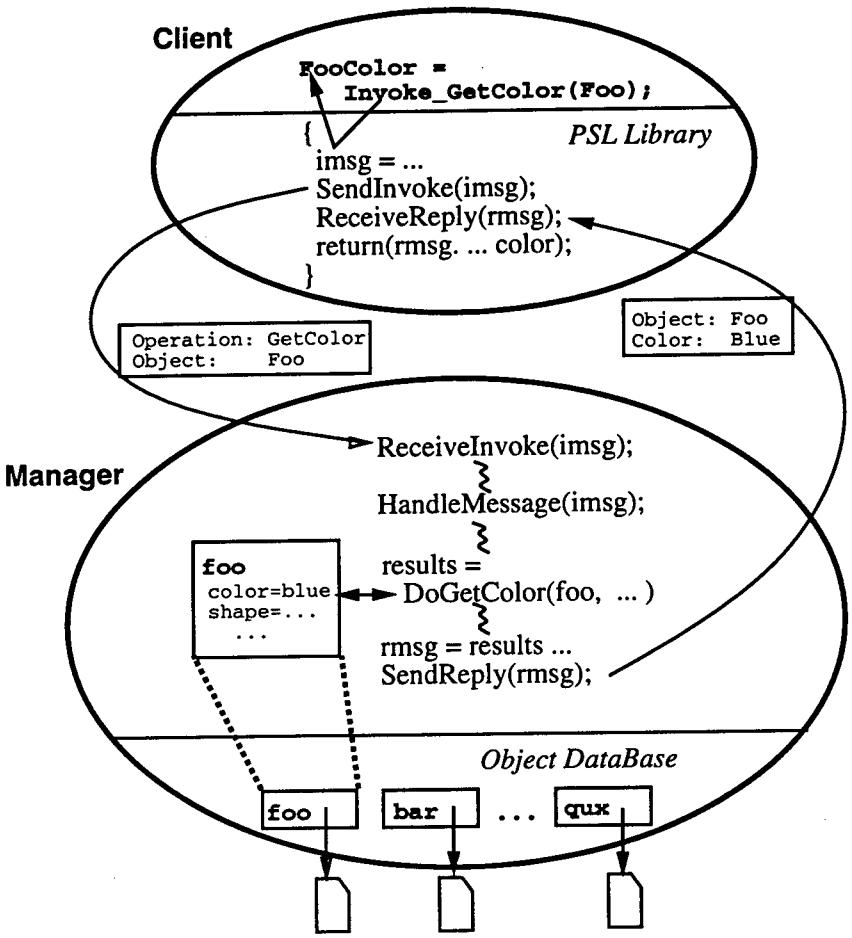

Figure 5.1: Corbus Object Request via Message

DoGetColor uses the message and the object data to compute the result of the operation, i.e., the object's color. (DoGetColor's access of the object data is illustrated as an arrow connecting it to the in-memory object data.) This data is copied to the operation's results, which are embedded in a reply message. The reply message is then sent to the client. (The reply message send is illustrated by the arrow from the manager to client, the message in a box by the arrow.) Back in the client's proxy code, the message is received, the operation results extracted, and returned as output data of the operation invocation call. At this point, the client execution returns to the client application code, which stores the results in FooColor, and then goes about its other business.

Figure 5.2 shows the shared-memory approach. As before, the client application code makes a call to the proxy interface in order to call the GetColor operation. Although the proxy interface used is the same as that of the message passing approach, the implementation is different. Instead of sending a message, the library code directly accesses the object data to perform the operation *in the client process*. In fact, the code used to perform it may be a very similar DoGetColor subroutine. As before, the manager has the representation data of the requested object; this data this data is used by DoGetColor. However, in this case, DoGetColor executes in the client and data access takes place in the client's memory. With Mach memory-management features, the manager's object data is memory-mapped into the client's address space. (The mapping is illustrated by a pair of dotted lines connecting the object data in the client with the object data in the manager.) As a result, DoGetColor can use in-memory object data in the same manner as the first case. Then, after DoGetColor computes the operation results, the results are returned as output data from the proxy to the client application code, just as if it had been extracted from a reply message.

5.3 Initial Memory Mapping

The above example of a client-side operation on a memory-mapped object is incomplete in that it assumes that the object is already mapped to the client's memory. However, in order for client-side object computation to work as described, the client (or rather the client's proxy library) must first contact the manager to set up the shared memory arrangement. This takes place by an exchange of messages with the manager, which happens

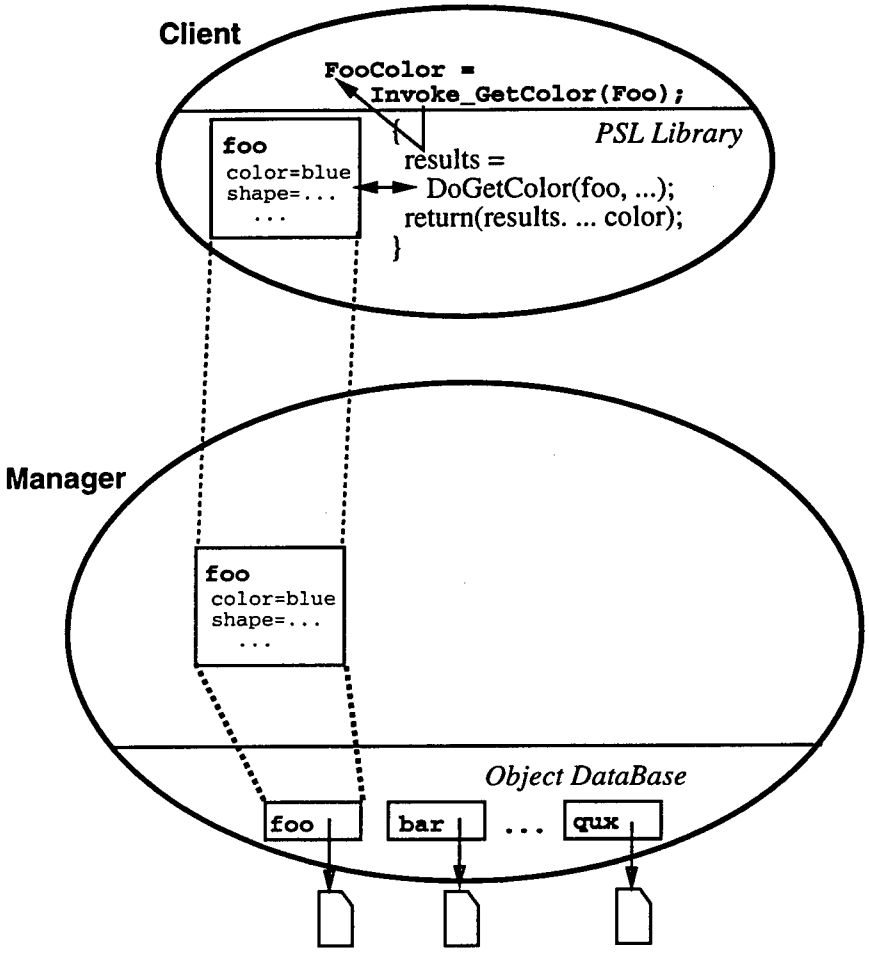

Figure 5.2: Corbus Object Request via Shared Memory

for each object the first time a client invokes an operation on the object. This communication facilitates the mapping into the client's address space of the shared memory, which is managed by the manager. Subsequently, any number of operations on that object can be processed by the client.

Each time a client makes the first use of an object, it makes an initial memory mapping. Therefore, each client's proxy code maintains a list of all the objects that it has already memory mapped. When a client performs an operation on an object, the proxy code first checks to see if it has been mapped. If so, then the proxy code proceeds with the operation. If not, then the proxy code sets up the mapping and operation proceeds.

To set up the memory mapping for an object, the client must contact the manager and request its cooperation. This request takes the form of an object request message for an operation we will call "Map" for the sake of exposition. The following list outlines the steps necessary for the first operation on a memory-mapped object.

1. Client application code invokes the GetColor operation on object Foo.
2. Client's proxy:
   (a) Can't find Foo in the list of memory mapped objects.
   (b) Invokes the Map operation on Foo, sending Map request message to the manager.
   (c) The manager:
      i. Receives Map message and checks for Foo in list of currently memory-mapped objects.
      ii. If the manager had previously created a port as the descriptor for Foo, that port it used; otherwise, a new port is created and associated with Foo.
      iii. Composes a reply message, which includes a port right for the Mach port that the manager uses as a handle for the memory associated with Foo.
      iv. Sends reply message.
   (d) Client's proxy receives manager's reply message.
   (e) Makes a kernel call to map the object, using the port right from the reply message; this causes the kernel to map into the client's address space an area which will hold Foo's object data.

(f) Adds new item in list of memory mapped objects and stores Foo and its memory address in the new item.

(g) Uses address to perform the GetColor operation on Foo.

(h) Returns results of operation as output of the call of GetColor.

3. Client application code uses results of operation.

When the kernel performs a memory-map request, it uses or creates an instance of a kernel construct called a *memory-object. Memory-object descriptor port* is the term given to the port that the client used in the memory-map kernel request. When the manager returns a memory-object descriptor port right to the client's proxy, the port may represent a memory-object which another client has already mapped. When two clients request an operation on the same Corbus object, the manager returns a memory-object descriptor port for the same memory-object. If this occurs, the manager is already managing the memory-object for the previous client; the new client will share that memory with the previous client. Because the manager implements a one-to-one mapping between Corbus objects and Mach memory-objects, it also implements sharing of object data between all clients that have memory-mapped an object. The result is that any change to the memory-object is immediately reflected in the address space of all the clients that have memory-mapped access to the object.

5.4 Memory Management

The above-described shared memory usage is invisible to the client application code, which simply calls an operation-invocation procedure and accepts the results. Below this interface, the client proxy code maps memory-objects and accesses the mapped memory to implement client-invoked operations. The client proxy's memory-object activities are limited to two interactions: first, requesting memory-object descriptor port from the manager; second, requesting the kernel to map the memory represented by the port. However, there is a whole range of additional memory management activity that is invisible to the client proxy code. A description of these other areas of activity should serve to highlight the remainder of the issues to be discussed.

In Mach, when a task maps a memory-object, a three-way relationship is established. There are three roles played by the task that maps to the

object. We refer to the first of these roles as the client[2]. The second of the three roles is played by another task, which we will refer to as the pager, which interacts with the client to give the client a send right to a Mach port. The pager retains the receive right to the port, so that any Mach messages sent over the port (using a send right to the port) will be received by the pager. Thus, the pager gives the client the capability to send it messages. However, in a memory management relationship, the client does not use the port to send messages. Rather, the client uses the port's send right as a parameter to the kernel interface function which it calls to map the memory represented by the port. As a result of receiving this port right, the kernel—the third player— can send messages to the pager. The kernel uses this port right to communicate with the pager to cooperatively manage the memory-object. Such kernel-pager interaction uses the Mach External Memory Management Interface (EMMI) protocol. Once the client has mapped the memory-object, it need have no further contact with the kernel or the pager. The kernel and the pager will use the EMMI to facilitate the client's use of the memory-object by direct memory access.

The first EMMI interaction occurs when the client performs a memory access at an address in the region of its address space into which the memory-object was mapped. Initially, little or none of the memory-object data is placed in memory for the client access. When the client accesses an address at which there is no data, the client performs a *page fault*, i.e., client's thread of execution traps to the kernel because there is no data for it to access. At this point, the kernel obtains the data for the page of memory that contains the faulted address.

The kernel's page data fetch is performed by sending a message over the memory-object representative port for the memory-object which contains the page with the faulted address. The pager receives the message, and replies to the kernel with a message containing the page data. The kernel places the page data in memory and resumes the client's thread of execution, which then accesses the data.

Other kernel-pager interactions are for other aspects of memory management. For example, the kernel may wish to eliminate the in-memory page

[2]We use the term "client" in this context because of the way memory-objects would be used in Corbus. The Corbus tasks that map memory-objects would be Corbus clients, which map a memory-object in order to access the data contained in a Corbus object. However, in Mach systems, any task can map a memory-object, irrespective of a client-server relationship. Thus, our use of "client" results from Corbus's client-server architecture and the use of Mach memory management within that architecture.

of one memory-object's data in order to make room for another. If the page has been modified, the kernel sends the page of data back to the pager, so that the pager may store it until it is fetched again. Such page flushes may occur not only when the kernel initiates a page-out, but also when a pager wishes to revoke a page for reasons of its own. Pagers may also push a page to the kernel to override the data currently in memory. Other EMMI interactions involve the management and enforcement of memory access modes on memory-objects.

5.5 Object Managers and Mach Pagers

This section explains the role of a Corbus object manager in EMMI. The kernel will demand memory-object page contents from some component acting as a pager. Because the memory-object's contents are the representation data of a Corbus object, those demands from the kernel must be met by a component that has access to the contents of a Corbus object. The natural candidate for the pager in this context, therefore, is the object manager itself.

A shared-memory Corbus object manager would act as a Mach external pager, maintaining a mapping between each memory-object that it pages and a Corbus object that it manages. When the manager receives page data request from the kernel, the manager determines which Corbus object corresponds to the memory-object in question and returns the object data. Since Corbus object data is stored by the manager in its Corbus Object Database (ODB), the manager must implement a correspondence from ODB data to the data associated with the Mach memory-object that it has associated with Corbus object.

However, there is still an open question as to where the pager/manager obtains the object data it needs to respond to a kernel demand for page data. At one level, the answer is that the manager gets the data from the ODB. However, there remains the lower-level question about the nature of the storage used by the ODB. The most straightforward approach to storage would be to use the services of a file server, either a standalone file server or a Unix server. This approach is workable, but has the drawback that every page data request would result in communication between the pager to the file server. Thus, this communication and the file server's processing would be inserted into the page fault handling loop.[3] For some kinds of object,

[3]The use of the Unix file service would entail the additional overhead that several file

however, this would provide an acceptable level of performance.

An obvious variant of this approach is for the pager/manager to cache object data in memory to avoid going to the file server for every page data request. This this raises the possibility that cached object-data would be in paged-out pages. A client page fault would result in a page data request to a pager/manager which would process the request by accessing paged-out cached object data. Then a second page fault would occur and the default pager, which manages the memory of most tasks, would page in the needed page for the pager/manager's use. Only then could the pager/manager respond to the kernel's original page data request for the client. Such nested page faults may offer better performance than the above approach and would require additional complexity for uncertain benefits.

A second approach would be for paging to be performed not by the object manager, but by another task. For example, it is fairly common on Mach systems to use external paging to provide memory-mapped file service. If object data were stored in memory mapped files, then the object manager need not retrieve data for memory-mapped object clients. Instead, the object manager simply sends the client a memory-object representative port for the file containing the object data for the object requested by the client. The client then uses the port to map a memory object containing the object data; the file server/pager performs the external paging. The manager itself stays entirely out of the loop in terms of memory-mapped object data access. However, the manager may have other duties, e.g., using the same file to access object data to fulfill object requests made via messaging by clients that do not access objects via memory-mapping.

This memory-mapped file approach, while feasible (assuming that a memory-mapped file service is available, such as that offered by some Unix servers), has a drawback that results from the Corbus ODB's storage of all data in a manager's objects in one file. It is probably not desirable to give every client memory-mapped access to all objects at one.

Another alternative is for the manager to use a different kind of pager, a very simple one that defines memory-objects that are backed to a paging store which is released when the memory-object is deallocated. When a client requests a memory-object representative port for a particular Corbus object, the manager would create such a memory-object, fill it with Corbus

service calls may be necessary to obtain the object data, due to the need to position the seek pointer. Other file services may be more convenient for pagers, if the file service interface allows the specification of an offset and extent to be retrieved.

object's data (obtained in a conventional manner from a file server), and return the representative port to the client. From then on, the pager handles all the memory management needed for the client's access to the object. Again, the object manager is out of the loop.

The primary drawback of this approach is that the Mach default pager, through a design oversight, does not provide the mechanism for tasks to manipulate memory-objects paged by the default pager. As a result, a separate external pager is required, though one that performs exactly the same sort of paging. Another drawback is that the entire Corbus object data must be obtained from the file server for insertion into the memory-object, even though the client may only access part of the object.

A final alternative is for the object manager to be the pager and in addition to managing the actual storage of object data. The ODB would still use a file service interface to manipulate object data, but this interface would *not* be implemented by communication with another server. Rather, the interface would be implemented by a library that is part of the manager itself. This library would implement the functionality of storing files on a raw disk partition device, and offer a simple interface for reading and writing from files. This would be the file service interface used by the ODB (though perhaps with an additional interface layer as discussed in Section 2.5.3).

In this last approach, the object manager would be acting very much like a memory-mapped file server because the manager actually manages the disk storage of objects. The difference would be that instead of implementing memory-mapped access to a simple object like a file, the object manager would implement memory-mapped access to whatever type of application-specific object(s) the manager supports.

The basic tradeoff of this final approach is between the costs of allocating a disk partition to every such manager, benefits of keeping paging, object management, and object data storage all in one task per application. The benefits are significant and probably worth the administrative overhead of having several tasks each managing a disk partition. There are obvious limits to the number of such pager/managers that there could be on a given host; the specific limit is based on the storage capacity of the host. However, these limits would not be substantially different than in the case where all the object managers on a host shared one file server. The main difference is that in the latter case, there is no need to apportion disk space on a per-application basis because all applications share the same large disk partition managed by the file server.

5.6 Memory-Object Data Representation

As described above, the major role of a Corbus object manager as a Mach pager is in relating ODB data to the page data requests that manager receives from the kernel. This section describes the approach to the relationship between ODB and page data. For the other approaches in which the manager is not a pager, there is a similar requirement for the manager to store object data in a way that the pager can relate ODB data to page data. For purposes of the simplest exposition, however, we describe the ODB-data/page-data management approach as it applies to an object manager that is also a pager. Note that the discussion makes no assumptions about how the ODB actually stores object data, e.g., via a file server or by managing a disk partition itself.

A Mach memory-object is a collection of pages of data that can be mapped into the virtual address space of potentially several Mach tasks. Once a client has mapped a memory-object, it has access to all the data. However, among all the pages in the client's address space that are allocated to a given memory-object, some pages may not be paged-in, i.e., backed by real memory. When a task attempts to reference a paged-out page of a memory-object, a page fault occurs. When the kernel services the page fault, it contacts the pager for the object to request the data for the specific faulted page of the memory-object. The kernel does not interpret the returned data in any way, so what the pager returns is entirely at its discretion.

Therefore, a Corbus object manager can represent an object in a Mach memory-object in any way it likes. So long as the manager and the client proxy code agree on how to interpret the data, the client's access to the object will work correctly. At a minimum, the manager must maintain a correspondence between each Mach memory-object it has created, and the Corbus object that it represents. Then, when the kernel sends a memory-object data request to the manager, it can extract the Corbus object's data from the ODB, locate the requested page within the data, and return that page of data to the kernel.

However, there remains the question of the relationship between the ODB data and the page of data returned to the kernel. One simple option would be the following: use the ODB in the usual way to put the object's data into its in-memory data structure form; compute the size of the structure and the requisite page offsets; and return a page of data as is. This would have the effect that the client would access data in the memory-object

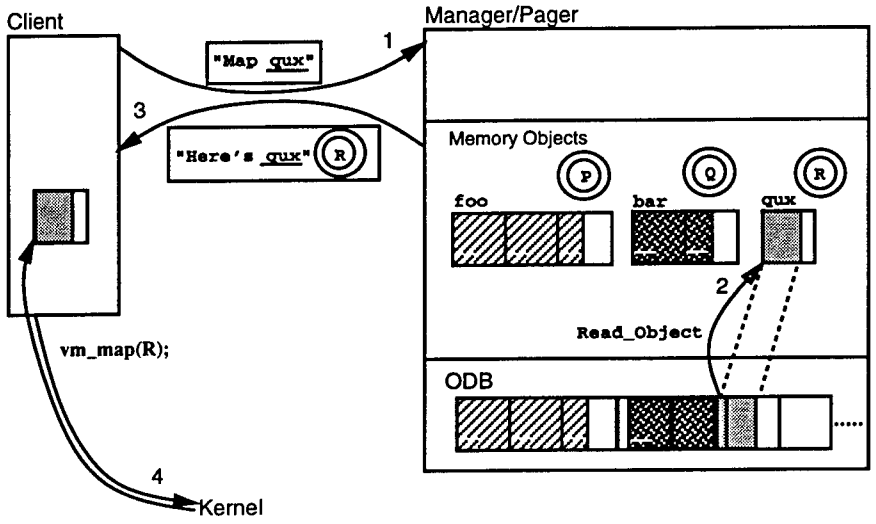

Figure 5.3: Pager Use of ODB

in the same form as the manager would. As a result, the code for client-side operations would be very similar to the code used to do operations in the manager in the usual message-passing approach.

This latter point is very significant for the process of manager generation. The manager developer typically writes a set of operation subroutines, each of which implements an operation using the in-memory data structure form of the object which was invoked. In this simple external paging model, the object data representation is the same in the shared-memory client as it would be in a manager which implemented the normal Corbus message-passing style of object access. As a result, the same kind of operation subroutines would be usable as is in client proxy code. This result is significant because it means that the paging model has no impact on manager software development.

Figure 5.3 more specifically illustrates the role of the ODB in external paging. The manager/pager is separated into three different parts: an upper part which handles messages traffic; a middle part which manages memory-objects; and a lower part which consists of the ODB software. The two lower layers show how each memory-object corresponds to one Corbus object; the data in the memory-object are the same as the object's data in the ODB.

Different objects' data are differentiated in the figure by different shading.

The critical difference between the two lower layers' management of object data is that in a memory-object the data starts at the beginning of a page. Pages are illustrated as a sequence of boxes of the same size. The data within the page are shaded, and the start of the data is placed on the left edge of the leftmost page. Thus, each memory-object's rightmost page has an unshaded right portion which illustrates a part of the page that is not used to contain object data (none of the objects happens to have data the size of which is an exact multiple of the page size).

In contrast to the memory-objects, the ODB object data is not laid out in accordance with page boundaries. The ODB layer illustrates the ODB data itself, which is shown as an array of storage in which various object's object data is laid out sequentially, with some gaps between objects to allow for growth of object data. In addition, the ODB storage is shown as divided into equal-size units of a page's worth of data. This division into page-size units is to illustrate the way that the page boundaries are *not* significant to ODB data layout. A single page's worth of ODB storage could contain object data more than one object, which is not the case with memory-objects.

The primary relation between the ODB object data and the memory-object is the act of reading data from the ODB and putting it in a memory-object. This act is illustrated by the "ReadObject" call to the ODB, shown as an arrow connecting the ODB data and the memory-object. A ReadObject call is made by the memory-object software, when it creates a new memory-object for a Corbus object. Although not illustrated, a corresponding WriteObject is also performed when the memory-object data is changed per client updates.

The final feature of the manager/pager part of the figure is the association between each memory-object and a Mach port which represents the memory-object. Hence, it also represents the Corbus object whose data is contained by the memory-object. Each port is illustrated as being named by a single letter enclosed in a double circle (reminiscent of a porthole on a ship).

The larger illustration shows a series of steps of the client-manager interactions involved in the client's receiving the memory-object corresponding to a Corbus object. The first step is a Mach message send shown by a arrow labeled "1" which goes from the client to the manager, and surmounted by rectangle which represents the client's message. The message is a request to memory-map a Corbus object, the name of which is "qux" which is simply an illustration of the Corbus UID that denotes the object. In the second

step, the manager receives the message and attempts to relate the name to a memory-object for the named Corbus object. Failing this, the manager creates a new memory-object for qux and performs a ReadObject for qux to get the data into the memory-object. This is represented by the arrow labeled "2" and "ReadObject" and going from the ODB to the memory-object. The third step is the manager's reply to the client, which is a message send like step one. The manager's reply message contains a port right for the port "R," which is the port associated with qux's memory-object. The fourth step is the client's use of R in a call to the microkernel to request that the associated memory-object be mapped. This is illustrated by the arrow labeled "4" and "vm_map(R)" and going from the client to the microkernel.

Subsequent unillustrated steps do not directly involve the client: interactions between the microkernel and the pager/manager to get the memory-object data and change the client's address space to contain it. Note that the ReadObject in step two is not really necessary. A particular manager may wish to supply all the page data of the object at the time it is memory-mapped in order to save the client some page faults. However, this is purely discretionary; if no data were read and map time, client reads would result in page faults to the kernel, page data requests from the kernel to the pager, and ReadObject by the pager to get page data.

The aspect of the ODB to discuss is the mechanisms employed when a client writes an object. Writes to memory-mapped Corbus objects are writes to the corresponding Mach memory-object. When the client updates the memory-object data, the data is eventually copied back to the pager by the kernel. Then, there is an inconsistency between the ODB data and the memory-object data shown in the middle layer of the manger. Therefore, the manager performs a WriteObject ODB call in order to store the new version of the memory-object data.

The WriteObject call simply uses the address of the memory-object data. The ODB code can determine the size of the object data and write only the memory-object data that comprises the object data, while ignoring the unused space on the last page of the memory-object. The placement in the ODB storage of the new object data is the responsibility of the ODB code, which does not need to be concerned with the page boundaries of the memory-object data. The object data can be moved within the ODB storage, e.g., if the object grew larger than its previous size, plus the gap between it and the next object in ODB storage. Just as the ODB code is ignorant of page boundaries, so so, too, is the memory-object code ignorant of ODB storage layout. The ODB code provides the memory-object code

with the object data without requiring the memory-object code to know anything except the name of the object. The pager needs to take no specific action to coordinate memory-object management with ODB management.

5.7 Client Object Sharing and Synchronization

In any shared-memory situation, some amount of synchronization is required to ensure that multiple writers do not interfere with one another by writing the same shared data. Computer science literature is replete with synchronization approaches that are equally applicable to synchronization among multiple threads in one process's address space and synchronization among multiple processes which share regions of real memory in separate process address spaces.

A simple approach to synchronization of shared memory is to implement a single mutual exclusion lock based on synchronization data stored in part of the memory-object itself. This is typically done in a fixed-size region at the start of the memory-object's memory. Any task which has access to the memory-object would also have access to the lock and can check to see if another task has acquired the lock. In addition, it can attempt to acquire the lock using mutual exclusion mechanisms of the kernel or hardware. Locking techniques such as [14] have been used in Mach systems [15] to implement common locking strategies such as a single-writer lock, where only one party at a time possesses the lock, and all parties write only when in possession of the lock.

A simple strategy like this prevents simultaneous writing (in multiprocessor systems) and conflicting sequential writing (in single processor systems), though at the cost of a single mutex for the whole object. More complex locking strategies can be implementing using the same techniques, with the difference that instead of having using a region of shared memory to hold data for one lock for the whole object, the region wholes data for several locks for different disjoint subsections of the memory-object. This approach is not as general as a single global lock because the definition of the subregions must take into account the data format of the memory-object. In Corbus terms, such a definition would be based on the application-type-specific data layout of an object's representation data. For example, with sequential block-structured objects, each of several locks could govern a (possibly dynamic-sized) sequence of blocks. In highly-structured record-like objects, each of several locks could governs a distinct field (or group

of fields). Finally, locks need not be single-writer locks, if it is appropriate (in terms of an application's requirements) to tolerate some amount of conflicting writing as the price of permitting some fixed number $N > 1$ of simultaneous writers.

In Corbus terms, any locking strategy would be implemented in application-specific proxy code. That is, the implementation of each specific operation would reflect whether the operation writes the object data and if so acquires the requisite lock. A minimal approach (in terms of application development) would be to provide a general single writer lock mechanism for all objects; application developers would need only to add a lock-acquisition call at the start of each write operation's implementation code. More complex approaches would involve either application-specific hand-crafted code, or the use of more complex autogeneration techniques based on IDL annotation which specify which fields or regions are required for a specific operation.

5.8 Replication

Object replication is a powerful feature of Corbus, and is related to shared-memory access to objects because such objects can be replicated. With object replication, there are multiple copies of an object, each of which is managed by a separate object manager on a separate host. Replication processing is performed as part of the computation of an operation on a replicated object. Replication processing is largely independent of the operation, and is usually carried out before the operation computation. Replication processing focuses primarily on the consistency of the object data which will be used to perform the operation.

Consistency refers to a comparison of the object data of the various replicas (as each copy of a replicated object is called). When an operation on a replicated object is requested, replication processing may determine that the current replica (the one to be used in the requested operation) does not contain the most recent version of the object. As a result, some some reconciliation processing may be necessary.

For replication of memory-mapped objects, the main replication issue focuses on the placement of replication processing in memory-mapped object access. The issue arises from the fact that object managers are not directly involved in operation computations on memory-mapped objects. In contrast, for objects that have a message-passing client-server interface, the

object manager carries out both the operation computation and the replication processing. Such managers cooperate to keep the replicas consistent and to propagate changes from one replica to another. For example, before a manager carries out a client's request, the manager may communicate with the managers of other replicas in an attempt to obtain the most up-to-date version of the object.

Therefore, in the message-passing approach, replication processing and operation processing are closely related areas of object manager functionality. However, in the memory-mapped approach, operation processing is performed in the client's proxy. There can be multiple instances of the proxy code operating on the single memory-object that contains the data of the shared object's replica. As a result, there is an issue as to when and where replication processing is to be done, and how it is to be related to operation processing. Shared-memory access to replicated objects requires an approach that allocates replication processing to either the client proxy, the manager, or some combination of the two.

5.8.1 Manager Replication Processing

One approach to shared-memory object replication is to leave replication processing in the manager, just as it is for objects with a message-passing interface to clients. In this approach, clients carry out operations, based on shared-memory access to object data. For replicated objects, however, clients would first request replication processing before proceeding with an operation.

The replication processing requests would take the form of exchanging messages with the manager of the replicated object. Clients would request replication processing for an object, and would wait for a reply from the manager. Once replication processing has been completed, the manager replies, the client receives the reply and then perform the operation on the object. Because of the blocking nature of client's requests for replication processing, the term "replication RPC" is used to refer to this client/server interchange. This replication RPC is exactly analogous to the way that a message-passing object manager makes a procedure call to perform replication processing before carrying out an operation. The difference in this shared-memory approach is that the procedure call is an RPC called in the client; but in both cases the object manager performs the replication processing.

Assessment of this approach can be clouded by a seeming contradiction.

If the goal of shared-memory objects is to minimize messaging between client and server, then this replication approach conflicts with shared-memory goals by adding more messaging between client and server. However, it is more accurate to say that the goal of shared-memory objects to lower the computational overhead client/server messaging; messaging itself isn't the focus, rather, messaging overhead is the focus.

A significant portion of messaging overhead involves the transportation of message data, including copying to and from client, server, and kernel address spaces. Shared memory eliminates the need to transport data between client and server to carry out an operation request because the client can access to the object data need for operations. Replication RPCs, however, need to carry a minimal amount of data, and so would impose little data transport overhead.

Another part of messaging overhead is the switches in execution context between client and server, sometimes including suspension of both client and server while another process executes. Mach thread migration techniques have virtually eliminated this overhead with an RPC mechanism that shifts execution from client to server and back again without a scheduling context switch. (See Section 7 for more details).

As a result of these factors, a replication RPC can approach a local procedure call in terms of overhead. Context switch overhead is reduced to the expense of one kernel interface call, while message transport overhead is reduced to the ID of the object which is to be replicated. Furthermore, even this overhead could be reduced if the object manager allocates a specific port for replication RPCs for each object. In that case, a client would not need to pass any data to the object manager in a replication RPC.

As a result of these optimizations, manager replication processing for memory-mapped objects compares favorably to the approach of objects with a message-passing interface. With manager replication of shared-memory objects, operation processing is carried out by separate processes linked by an efficient RPC mechanism. This RPC mechanism need not be prohibitively higher in overhead than the procedural-call interface between replication processing and operation processing done in the same process (either a message-passing manager doing both or a memory-mapping client doing both). The additional cost of a replication RPC (as compared with an actual procedure call) is small enough to justify the combined use of replication and memory-mapped access, without undermining the benefits of either.

5.8.2 Client Replication Processing

Although it is feasible to split replication processing and operation processing client and manager, it seems worthwhile to examine an approach in which a memory-mapping client does both replication and operation processing. In this approach, a client would assume the object manager's role in coordinating the consistency of the current replica with other replicas. However, only one client at a time could be the coordinator for that replica, in order to maintain the Corbus replication mechanism in which each replica has one coordinator. Therefore, different processes would be coordinating replication for a specific replica at different times. As a result, several complicating factors would be have be handled.

First, the rotation of coordinator responsibilities would make it challenging for *other* replica's coordinators to determine the current coordinator of a given replica. Such determination would require the maintenance of some state data that identifies the current coordinator. In addition, there would have to be a mechanism for other coordinators to access this data.

Second, each client would have to be careful about relinquishing its coordinator role. Even if a client had completed its use of an object, it might still be performing replication activities initiated by the coordinator of some other replica of that replicated object. Such overlapping requests could delay relinquishment of the coordinator role, thus delaying other clients from performing operations.

A third complexity factor is the need to maintain the state of replication processing so that the state can be transported with the coordinator role. Such a mechanism might use forwarding of replication requests (from other replica's coordinators) or binding outstanding requests with the token to be acquired by another client to become the replica's coordinators. This would require additional computational overhead, state data management, and inter-client protocols.

There are more complicating factors that could arise from yet other alternatives, e.g., approaches in which an object's manager and one client at a time share the coordinator role. The above three complicating factors, however, should demonstrate that the additional functional requirements and computational overhead lead to significant overhead. The costs are likely to outweigh the relatively modest cost that shared-memory manager replication has in comparison to the basic Corbus approach (manager processing of replication of objects with message-passing interfaces).

There is another complexity factor in addition to those derived from the

requirement for the current coordinator client to work with other coordinators. This additional complexity is simply the work required for the clients of one replica to ensure only client at a time acts as coordinator. At least two locks would be required (one for coordinator of a replica, and one for a coordinator that can propagate updates), as well as additional state information to prevent a new coordinator from repeating the replication processing just completed by the previous coordinator.

Network communication is the final factor in comparing of the client-coordinator approach with the manager-coordinator approach. The previous section has established that manager coordination imposes only modest overhead on shared-memory object access. Attempts to minimize this overhead (either by client coordination or other schemes) overlook the fact that overhead of client/manager communication is often a minor component of the overhead of replication processing. In many cases, the bulk of overhead results from network communication with other replicas' managers and the time required for the replication computation performed by them. As a result, even if considerable complexity achieved a reduction in the client/server component of replication processing, this reduction would be small when compared with replication processing as a whole.

Therefore, this client approach to replication processing, while not impossible, adds additional overhead the same as the manager approach. does share with the manager approach some additional overhead. The manager approach, having the virtue of simplicity (the replication RPC being the only new aspect), seems superior.

5.8.3 Read Operations

Read operations require replication processing to determine whether the replica to be read from actually contains the most recent version of the replicated object's data. The number of other replicas to be consulted can range from zero to all other replicas, depending on the amount of risk of outdated reads that the application can tolerate. An alternative approach to replication is made possible by this use of replication processing for read operations.

For comparison, consider the way that a shared-memory object read operation would include replication in the manager-coordinating approach. The client's proxy code, before performing the read operation on shared memory, would perform a replication RPC to the object manager. The manager would contact the managers of other replicas and if necessary, would

obtain current object data from them and update the local replica's data. For any page of the corresponding memory-object that is affected by such updates, the manager/pager would "push" the page, i.e., request the kernel to refresh the page contents with newly-provided page data. The manager would then return from the RPC and the client would resume processing with a fresh copy of the object data.

A second approach combines read-operation replication processing with the paging functionality of the manager/pager. Replication processing would not be done for every client request but rather, for every page request. This approach would change the semantics of replication, but could be an alternative mechanism beneficial for some applications. The different semantics result from the different breakdown of responsibilities in the shared-memory approach: object requests are handled by client proxies rather than managers, while managers maintain the object data and perform replication processing. In this approach, the manager could perform replication processing for every page data request from the kernel. For each object request, the manager would identify the managers of other replicas of the object, and would engage in network communication with other replicas' managers before replying to the kernel's request. Any required object data updates would be performed, and the page data then returned to the kernel.

A more likely approach would be for the manager to reply immediately to the kernel, thus avoiding delays in handling until the completion of network communication with other replica's managers. In this alternative, the manager would engage in replication processing *after* servicing the page data request. If the requested page were affected by an update resulting from replication processing, then the read operation would have proceeded with out-of-date data, but subsequent read operations would use the updated data.

However, this approach would not be suitable for all applications, for example, those with frequent write operations. Frequent writes would result in frequent out-of-band updates which might effect a read operation whose execution caused the page fault. As a result, the pager would always be trying to catch up on the updates, and the clients are always seeing out-of-date data. Therefore, this second approach may serve some applications very well, but it may not be optimal for others.

A final note about this alternative approach concerns the frequency of updates. Updates are entirely dependent on page data requests because replication processing is initiated only by page data requests. It is possible for all of the pages of an object to remain resident for an arbitrary amount of

time, so that a significant time passes before a page-out and subsequent page-fault causes replication processing for the object. As a result, several read operations may occur on out of date object data. To ameliorate the potential for such behavior, the manager could undertake replication processing at other times, and in a way that is related to client operations. A proxy's execution of a client request could signal the manager in some way, so that the manager could perform some replication processing to ensure that the local replica isn't out-of-date. This signal could either be a message (an asynchronous one that the client wouldn't wait on), or an increment of an event counter in the same shared memory region used to hold lock data. In the former case, the manager would have to listen for such messages. In the latter case, the manager would have to map the memory and periodically check it.

Use of this page-based approach to replication processing is consistent with the standard operation-based approach. Both approaches could be implemented by Corbus software, either in server library code, or library code used by IDL-generated client code for shared-memory object access. Either approach could used on a per-application basis, depending on which approach is deemed more valuable by the application developer.

5.8.4 Write Operations

Note that sensitivity to read operations on out-of-date data application-specific. Some applications may tolerate some amount of reading an out-of-date replica in return for not having to delay read operations with a replication RPC. Write operations are fundamentally different. Clients cannot forego replication processing. If a client failed to make a replication RPC before writing the data in its memory, then it might update an object, thus resulting in inconsistent replicas. Part of the function of replication processing of write operations is to prevent such inconsistent updates. An irreconcilable set of replicas results from inconsistent updates that could occur if clients of all replicas skipped making a replication RPC, deferring replication processing the object's manager to perform at a later time when the written pages are flushed back from the kernel.

Another significant difference between read and write operations is that write operation replication processing entails two additional aspects. As with read operation replication processing, the manager communicates with some or all of the other replicas' managers in an attempt to obtain the current value of the object data. With write operations, this communication

also establishes a lock, so that the other contacted managers won't initiate additional write operations on the object until the current one is finished. Another aspect of write operations is a second round of communication with the other replica's managers. This second round propagates the newly-written data, and releases the lock.

The following is the sequence of steps for a write operation on a replicated memory-mapped object.

1. Client proxy receives procedure call for operation on object.
2. Client proxy makes replication RPC to manager.
3. Manager obtains lock and perhaps up-to-date object data.
4. Client proxy performs write operation.
5. Client proxy makes a second replication RPC to manager.
6. Manager pulls the pages of the object that have been written.
7. Manager releases lock and propagates new object data.
8. Manager stores new version of object data using pulled pages.
9. Client proxy returns operation results to client software.

This list of steps shows two main features. First, some replication processing must be done before the operation, in order to ensure proper locking with other replicas. Second, some replication must be done after the operation, to propagate the updated object data.

An additional factor relates to pulling of effected pages in step 7. It is possible that the kernel will push dirty pages to the manager/pager. If pushed pages are for an object for which the manager has not obtained a lock from other replicas, then the push resulted from a misbehaving client that performed a write operation without making a replication RPC. In such cases, the manager could treat these pushed pages as a result of write operations, by obtaining a lock and propagating the values. However, if the client wrote and out-of-date version, then an inconsistent replica results. Therefore, the manager must drop the pushed pages and invalidate the data resulting from client's unlocked write operation.

5.9 Paging and Data Propagation

A more fundamental consideration for replication is the amount of messaging done between the manager/pager and the kernel. For both read and write operations, the manager may find a newer version of the object in another replica. In such cases, the pager would push pages of new data to the kernel. It may be that the amount such kernel/pager messaging would be more than the amount of messaging done with message-passing interface between client and object manager. Therefore, an application-specific comparison of messaging overhead is necessary for analyzing the tradeoffs between message-passing and shared-memory approaches to replicated objects.

The key aspects for such tradeoffs assessments are: the degree of read consistency required, the degree of write consistency required, and the likelihood of replication processing finding that the local replica is out of date. In cases where write operations are more frequent, most operations will already use replicas with current versions, because the local replica was involved in the last write operation and hence had the latest object data propagated to it. Additionally, background update mechanisms may execute frequently enough with respect to the frequency of operations so that few operations encounter out-of-date replicas.

Nevertheless, there may be applications in which a significant proportion of operations on replicated objects encounter out-of-date replicas. As a result, many operations would require replication processing to update pages with newer object data. For such applications, the use of both replication and memory-mapping may not be useful. In other cases, there may be applications for which there is a low frequency of requests requiring updates before operation processing, but for which but update propagation *after* write operations is a significant expense. That is, for write operations, the pager must pull dirty pages in order to get the new object data to propagate to other replicas. In some cases the amount of kernel/pager messaging might outweigh the amount of messaging that would be done with a message-passing interface. For such applications, both replication and memory-mapping also may not be useful; alternatively, a hybrid approach could be beneficial. In such a hybrid approach to shared-memory access to replicated objects, read operations would use memory-mapped data, while write operations would be done via a message-passing interface.

However, such judgements must also take into account the distributed environment in which the replication happens. Some replicas may be stored

on hosts that have high communication costs with respect to other replica's hosts, in terms of limited bandwidth, latency, unreliability, etc. In such environments, the cost of local messaging for pushing and pulling pages may be far outweighed by network communication costs for replication processing. Therefore, it may be that for one group of operations, shared-memory access is wins over message-passing for operations which require no updates (or small updates), while for another class of operations the extra local message overhead is not a significant component of the cost of the operation. For such applications, it may be sensible to retain a shared-memory approach because the benefit to first class of operations is significant while the detriment to the second class is dwarfed by other detrimental factors.

Chapter 6

Remote Object Data Caching

6.1 Introduction

Remote object data caching occurs when object data is shared between a client and server that reside on different host. Such sharing is particularly beneficial for applications which use objects that are large, or extensible, or have serial or array-like structures. For example, file data caching is a critical element of network and distributed file systems such as NFS and Sprite. In terms of benefit to distributed applications, remote object data caching has similar results as local shared memory between client and object manager. Object data is available to client proxy software to access directly, rather than depending on messaging for operation requests and replies between client and manager. That is, both techniques share the same goal of optimization of message traffic and local access to object data. Just as distributed messaging is more costly than local messaging, reduction of the distributed messaging overhead has a greater benefit in distributed applications.

This chapter describes how remote object data caching techniques can be implemented in Corbus, so that applications can reap the benefits without having to be aware of the mechanisms involved. Two basic approaches are compared, one of which takes advantage of Mach's external memory management facilities. More details and tradeoffs concerning the Mach approach are also described. In particular, requirements for distributed data consistency are discussed, as are options for handling heterogeneity between

clients and object managers.

6.2 Data Caching in a Distributed Environment

The basic concept with object data caching is that some or all object data resides in the client address space, so that the client proxy code can directly access the data in order to implement client object requests. The local shared-memory approach described in Section 5 is a particular instance of this approach. By taking advantage of locality of client and object manager, the management of the cache (containing the entire object) can be done by the object manager acting as a Mach pager.

In a distributed environment where clients and object managers may be on different hosts, some kind of network communication is necessary to manage client-side object data caches. Object data must be sent from manager to client. Client updates must be sent back to the manager and also propagated to other clients of the same object. Client updates must also be coordinated to prevent the multiple simultaneous updates from creating inconsistent versions of the object data.

The most common approach to remote data caching is joint management of the cache by client proxy code and manager code. Many network and distributed file systems use some variant of this technique. When client requires file data, the client proxy communicates with the remote file server, obtains the data, and caches it in memory for later use. If the same data is accessed before it is flushed from the cache, then the subsequent access requires no communication because the data is already in the client's memory. If the client's access includes writing the data, the client proxy first obtains a single-writer lock from the file server; when the writing is complete, the updated data is sent back to the file server along with the lock release. The file server then updates the caches of other clients accessing the file.

This client cache management approach has two drawbacks that can be addressed in a Mach system. First, the client/manager interaction requires a cache management protocol. While such protocols are relatively simple in distributed file systems, the generalization is difficult when this technique is employed in a reusable way for distributed object applications. The protocol would have to take into account the application-specific structure of object data. This generalization of file caching for arbitrary types of application data poses real challenges for IDL-based software generation tools. In order to obviate this difficulty, it may be simpler to cache each object's data in its

entirety, so as to not have to generate code to fetch and cache portions of objects. While simpler, this may incur unnecessary overhead resulting from obtaining object data that is never used.

A second drawback is that cache data can be paged out because the management of the cache as object data (performed by application software) is uncoordinated with management of the cache as data in the client's memory. As a result, paging overhead can be part of the cost of a cache search, and even after a successful search the required data may still need to be paged in. This potential for undesirable paging activity is a result of making client software perform cache management.

The Mach memory management approach to cache management entirely removes cache management responsibility from the client, and unifies cache management with memory management. Object data is cached in client memory that is paged by the object manager, just as in the local shared memory case described in Section 5. Object data can be obtained on a page-by-page basis (rather than having to be obtained for the entire object), and this can be done without any knowledge of the application-specific structure of the data. Paging and caching are combined. That is, a given page of object data is in cache if it is paged in, and out of cache if not. Thus, cache hits are synonymous with ordinary memory access, and cache misses are synonymous with page faults. Pages of data are only obtained as needed.

While this unification of functionality is simpler than the file system caching approach, it still shares the same basic requirement for consistency of object data in the face of potentially multiple conflicting writers on different hosts. This consistency requirement is one of the two principle differences between paged remote object data caching, and local shared memory of object data. The other principle difference is that in the former case, the pager/manager is on a separate host from the client and the kernel-pager communication takes place via distributed rather than local IPC.

Another significant factor with this approach is the heterogeneity of client and object manager hosts. An object-based generalization of the file caching approach can deal with heterogeneity fairly easily. Manager and client proxy exchange object data of entire objects at a time. Despite other drawbacks of this approach, a benefit is that it allows the object data to be encoded in a canonical type by the sender, and converted to local format before being placed in cache (when the client is the receiver) or object storage (when the manager is the receiver).

In the Mach paging approach, however, the Mach kernel is the receiver of messages from the object manager. The kernel has no facility for de-

coding application-specific canonical types. Rather, the kernel simply takes page data from the pager/manager, and places the data as is into the client virtual address space. As a result, general support for heterogeneity in the paging approach requires more complexity of mechanism. As the following sections show, this complexity may not be warranted, especially when considering that heterogeneity factors include differences in hardware, operating systems, page size, word size, byte ordering, and subtler differences in data layout resulting from different compilers or linkers.

6.3 Consistency Management

This section describes the need for consistency management in the Mach paging approach to remote object data caching, and means to address this need. The similarities and differences between cache management and local shared memory are stressed, since these drive the need for additional requirements beyond those of local Mach paging.

6.3.1 Local vs. Distributed Consistency

In any shared-memory scheme, data consistency and coordination are important issues. Mach's external paging mechanism automatically provides data consistency when a pager and its clients reside on the same host. This results from the fact that when multiple clients map the same memory object, each client's virtual addresses for that object map to the same physical addresses. Thus, a change to a memory object is immediately reflected in the memory of every client that has mapped the object. No other propagation of updates is needed. This direct sharing requires some coordination of writes to a memory object if there are multiple writers and if collisions are undesirable. Such coordination mechanisms are fairly straightforward in the case when a pager and its clients are on the same host.

In a distributed system, a pager may have clients on different hosts. For distributed clients, local consistency and coordination mechanisms are insufficient. Clients on different hosts lack this local memory sharing of clients on the same host. As a result, when a client on one host updates mapped data, the clients on the other host do not automatically see the updated data. Thus, two different hosts may have inconsistent versions of the same memory-object which contains the data of some Corbus object. Some form of data propagation is required to ensure that each client's version of object

data is consistent with other clients' versions of the data. Therefore, the distributed use of external paging in Corbus/Mach requires new functionality for distributed consistency and coordination of memory object access.

Propagating the new page data is the principle action that a pager must take in response to a page write-back. This process is illustrated in Figure 6.1. When a pager receives a write-back message from the kernel on one host, the pager must determine if there are other kernels that have a copy of the memory-object referenced by the write-back message. If so, then these kernels' versions of the page data is out-of-date and the pager must send the newly written-back page data to them.

In Figure 6.1, two hosts are shown, with the pager residing on Host A. There are three clients of the pager, two on the pager's host and one on Host B. The pager holds the data for a memory object that is schematically named foo. Foo's data is mapped into the address space of each client via a dialogue between the pager and the kernel. Since clients exist on two machines, the pager communicates with each machine's kernel. Each kernel arranges for the memory object's data to be in each client's address space. This memory mapping is illustrated by dashed lines between each client's virtual address space containing the memory object and the kernel-managed real memory.

Initially, the three clients have the same view of foo's data. Then, assume that one of Host A's clients writes part of foo's data in its own address space. Because the clients on Host A share the same real memory, this write is visible in the writer's address space and in the address space of the other Host A client. However, the client on Host B is ignorant of the change to foo until the pager propagates the new data via communication with Host B's kernel.

This simple approach to propagation ensures that changes to a memory-object on one host are propagated to other hosts. However, this approach is insufficient for handling a whole class of potential collisions. The reason for this inadequacy stems from the fact that several kernels may be send messages to a pager and the order in which the pager receives them influences the way in which the page data is stored.

For example, suppose two clients on different hosts have mapped the same object and write to the same page. Suppose that Client A's kernel writes back the page first, the pager receives it, and propagates this value to Client B's kernel, thus nullifying the write made by the second client. Alternatively, the Client B's kernel may have sent a write-back message after the first kernel does, but before the pager handled the first kernel's write-

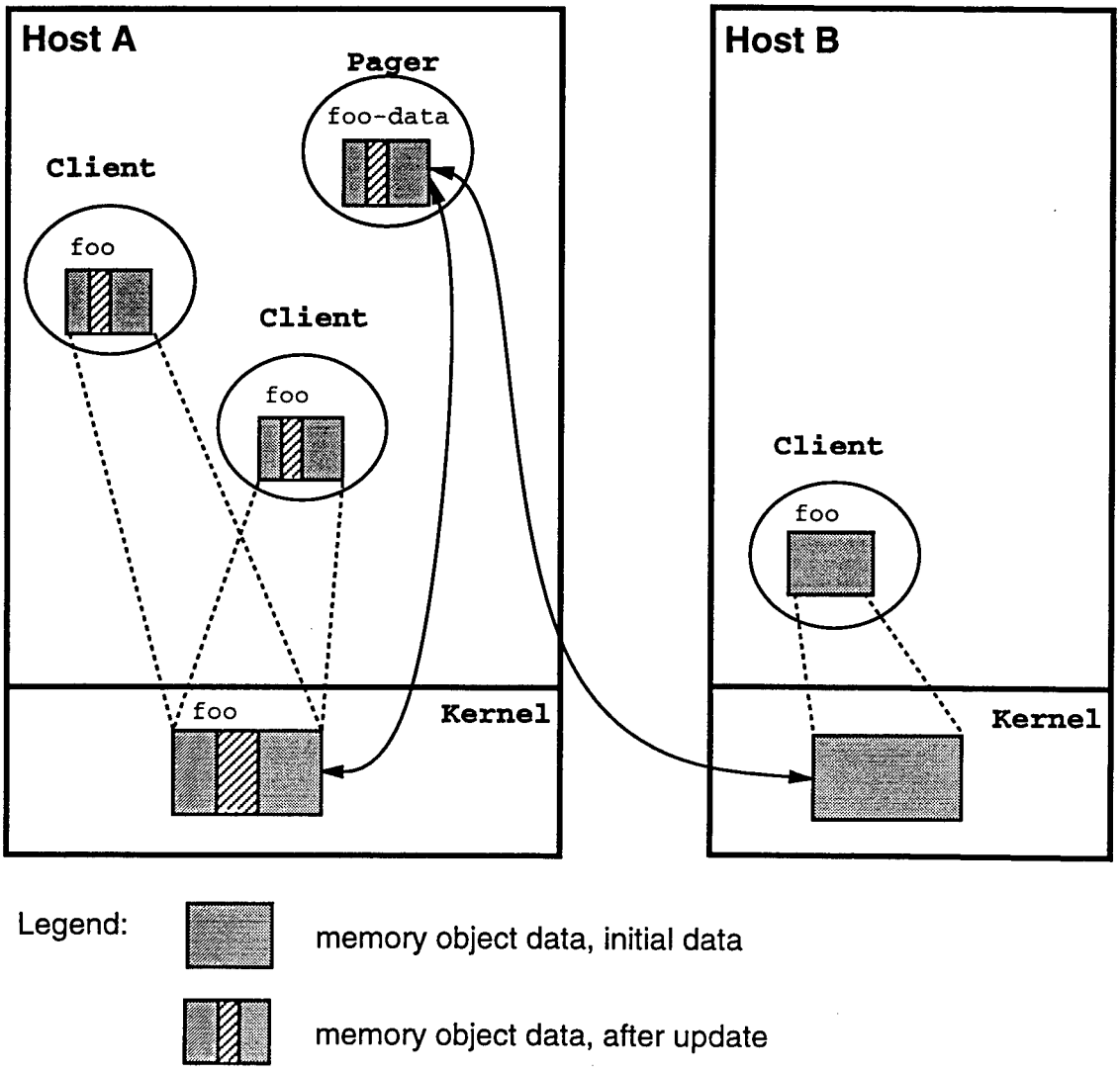

Figure 6.1: Pager Propagation of Dirty Pages

back message. In this case, the pager will first handle Client A's version and then Client B's version. On Host B, the data first gets changed from Client A's version to Client B's version. It is then changed back to Client A's.

6.3.2 Potential Solutions

There are a variety of ways to do locking as part of consistency processing, regardless of the type of environment. However, the distributed case is unique for two reasons. First, when two parties share local real memory, a lock can be considered representative of shared data. This is not the case with distributed shared memory; one must lock a segment of memory before it can be shared. This amounts to a Catch-22 problem. Therefore, some fundamental part of the locking scheme involves message passing between a client and a server. Second, the distributed case requires that newly-updated object data be propagated among all of the pertinent servers lest a client perform an operation based on out-of-date data. Hence, there must be a way to identify what has been changed upon object updates and then those changes must be propagated throughout the system. Note that these actions would occur after a lock has been released by a client.

There are many different approaches to pushing and/or pulling changes in conjunction with locking. The literature of distributed shared memory (DSM) techniques is replete with various approachs. Though developed originally in the context of a DSM abstraction for parallel programming using multiple cooperating networked hosts, the DSM techniques apply just as well to object consistency. That is, the memory containing an object's data can be treated as a DSM space spanning the various hosts which have a client accessing the object. IDL tools would generate software to use one of these DSM approaches to consistency. More variety in consistency approach can derive from more sophisticated IDL tools, i.e., multiple possible approaches could be offered as alternatives to application developers.

Approaches to consistency may also vary with regard to which system component will actually do the push/pull of changes, e.g., Mach in object pager/manager. If the Mach pager/manager manages the lock, then it can distribute object data changes when a client releases that lock. Reference Figure 6.1 Data Propagation, for a example of how this might be done. Clients obtain a lock from the pager/manager via messaging, and only then perform the update. After completing the update the client releases the lock, again via a message to the remote pager/manager, the manager propagates the changes by pulling the dirty pages from the client's kernel, and

pushing them to the kernels of other clients. A single-writer lock is the simplest approach to locking, though several alternative locking approaches are presented in Section 5.7.

6.4 Caching in Heterogeneous Environments

In a heterogeneous environment, the client and manager can exist on different hosts, which can have different hardware architectures. Heterogeneity of hosts is a significant complicating factor in distributed object cache management via Mach external paging. The approach described above works well in cases where the client and object hosts are homogeneous, because the data is transmitted in the form native to the common host type. The various benefits of this approach result in part from the lack of involvement of the client in cache management.

However, this same lack of client involvement creates difficulties for cases of heterogeneity of client host and object host. Transmission of object data between client host and object host must be able to take into account potential differences between the two hosts. In these cases, object data must be transmitted in the form of host-independent canonical data encodings, or cantypes, because the client and manager would otherwise interpret the object data differently. Reception of cantyped object data is performed by the kernel, rather than client library software. However, the client can include application-specific library software for converting canonical data, while the Mach kernel cannot. The Mach kernel can only take the page data supplied by the pager, and write it into the client's memory. If the pager supplies data in a canonical format, then the data will remain in canonical format in memory, rather than being translated. Therefore, new techniques must be applied to the management of canonical data in distributed Mach memory management.

The following sections describe some techniques for managing cantyped object data. Although there are a variety of issues, the basic tradeoff is fairly straightforward. There are several alternative approachs, each with positive and negative points:

- Benefit from a memory-mapped object data cache without the cost of cantypes, but be restricted to support for homogeneous cases.
- Benefit from a memory-mapped object data cache, but pay in increased complexity and deceased performance by always transmitting

data in cantype format.

- Transmit data in cantype format only when needed, and increase performance in some cases while increasing overall complexity.
- Forego the benefits of memory-mapped caching in favor of the simplicity of client-library cache management and cantype translation.

In each case, there is a tradeoff to be made. Subsequent discussion of design approaches should provide more information that is useful for making an informed decision among the various choices.

6.4.1 Manager Maintenance of Cantype Data

Cantype management is the main source of complex requirements for pager/managers to efficiently perform object data caching in a heterogeneous environment. There are inherently high costs associated with a client's use of a memory-object containing cantype data. The cost arises from the fact the cantype format of the shared data differs from the native format used by the operation subroutines that actually perform the operations on the object data.

When object data is transmitted in cantype form, there is an extra step in a manager's response to a memory-object data request. After the manager gets the data from the ODB and puts it into a data structure, it must convert the data structure into its cantype representation. It is this octet-sequence, rather than the byte sequence of the data structure, from which the manager draws the page of data that is returned to the kernel. Thus, object data exists in three versions: the actual ODB data, the in-memory representation of the ODB data, and the in-memory cantype version of the object data. These must managed jointly.

Clearly, there are performance issues associated with this approach. The main tradeoff involves object updates. If a manager maintains an in-memory copy of an object's cantype value, then the corresponding ODB data will be out of synch when the in-memory cantype value is updated. As a result, the manager must maintain consistency between the ODB object data and the in-memory cantype object value. This maintenance could be done using either one of two approaches. The first employs a consistency check for each memory-object data request. The second requires a consistency update for each update of the ODB value. The consistency check is a comparison of the object version between the ODB value and the cantype value. In cases of a

mismatch, the cantype value would be recomputed based on the new ODB value. The consistency update checks to determine if there is an in-memory cantype value for the object being updated. If so, the cantype value would be recomputed.

The simplest approach is to eliminate maintenance of the in-memory cantype version, by eliminating this version. The manager would perform this ODB-access and data-conversion for every memory-object data request and every memory-object data-flush. If objects of the type tend to be small, e.g., less than a page after conversion, then the cost of the lookup and conversion may be sufficiently low, thereby justifying the additional work of maintaining an in-memory cantype copy. For larger objects it may be desirable to save the results of the ODB-access and data-conversion for later use in order to avoid redoing large ODB-accesses and conversions.

The usefulness of such functionality is dependent on the relative frequency of Corbus object update operations versus memory-object data requests. The former is governed solely by client behavior, but the latter also depends on paging behavior. A memory-object data request occurs only if a client tries to access an object. This can be repeated if the required page gets paged out before the next access to the object.

It appears that there is no single approach to optimizing the processing necessary to use cantype data in external paging. One cannot extrapolate from expected client behavior to a useful approach due to the role of paging. Therefore, the best approach is for the manager to be adaptive. For each object, the manager should note the frequencies of update and page requests, and maintain the consistency of in-memory cantype values for objects with a high page request rate and a low update rate.

A relatively small amount of work would be required to track object updates and page requests. Also, the handling of each page request would include a lookup of the object in a cache of cantype values. This extra effort may be worth the benefit of an approach that avoids excessive computation of cantype values. However, it is conceivable that for some types of objects and client access patterns, this effort would result in little additional efficiency. Hence, from a development point of view, the manager code should perform cantype value computation for every page request until performance issues justify use of a more sophisticated technique.

6.4.2 Homogeneity Optimization

It is possible for a pager/manager to differentiate between, and provide service to, heterogeneous clients and homogeneous clients. (Mechanisms for performing the differentiation are described in Section 4.2.1.) The benefit applies to cases where the client and manager are on homogeneous hosts, and therefore can forgo conversion to cantypes by the manager and conversion from cantypes by the client. However, doing so entails a non-trivial level of complexity. The complexity stems from the requirement for the pager to maintain two different memory-objects for each Corbus object.

One memory-object would contain a cantype representation of the Corbus object data, while the other would contain a native representation of the Corbus object data. For heterogeneous clients, the pager would provide a memory-object with cantype data, while for homogeneous clients the pager would provide a memory-object with native representation data. Heterogeneous clients accessing a given object would share the same cantype-based memory-object, and homogeneous clients would share the native object. However, clients of a given Corbus object would no longer share the same memory-object that contains the Corbus object's data. Because there could be two memory-objects for one Corbus object, the two memory-objects and the ODB would need to be synchronized.

This combined approach would not be significantly impacted by a change to Corbus whereby the ODB stores object data in cantype format. Such a change would relieve the manager of the burden of converting object data to cantype format. However, the result would be increased complexity. To avoid requiring clients on the same type of host to access the object data in cantype format, the manager must convert from the ODB cantype format to the native format for a homogeneous client. Thus, the manager would still have to convert cantype and native representation types, but the conversion would be optimized.

6.4.3 Client Processing of Cantypes

As the foregoing shows, there are some complex requirements for pager/managers to efficiently perform cache management in a heterogeneous environment. If this complexity made matters simple for clients, then the manager's extra work might well be worthwhile. Unfortunately, this is not so, because there are inherently high costs for a client using a memory-object containing cantype data.

The cost arises from the fact the cantype format of the shared data differs from the native format used by the operation subroutines that actually perform the operations on the object data. In order to use the usual kind of operation subroutines, the shared-memory cantype object data must be copied to native-representation structure.

An alternative approach is to use the cantype data as is. To do so, a new and different kind of operation subroutine would be needed— resulting in a significant change to the manager development methodology. Even with such a change, conversion would still occur in order to get and set parts of object data using the native representation values specified by the programming language used. Note that such partial translations may need to be supported by new autogenerated code which can traverse the entire cantype form in order to find the required field. Also, partial translations can be problematic when a write to a field causes the size of the cantype data to increase.

Therefore, there is some unavoidable cost. To better understand this cost, consider the comparison with a normal Corbus manager that provides message-passing access to objects. When a client invokes an operation, the client proxy code converts the operation parameters to cantype format, and includes them in the message which it sends to the manager. For the manager to perform the requested operation, it must convert these cantype parameters to the manager's native representation (which may be different from the client's native representation) in which the invoked-upon object's data is maintained, and which the operation subroutine uses. Likewise, output parameters must be converted to cantype format by the manager before sending the reply message, and client proxy code must convert back from cantype format before returning the operation results to the client code. By contrast, the situation is reversed in a shared-memory approach. The operation parameters, specified in native format, do not need to be converted to cantype format because the operation is performed locally in the proxy code. However, the object data comes from the manager in cantype format and must be converted to the native format for use with the parameters by the operations subroutine.

Therefore, some amount of conversion to and from cantypes is necessary in heterogeneous cases whether the manager uses a message-passing approach or a shared-memory approach. However, there are two critical differences. First, the entire set of object data must be converted for the shared-memory approach, rather than the parameter data. In many cases, the object data is larger than the parameter data of most or all operations

provided by the manager. Thus, there is the requirement for what is likely to be more computation for cantype conversions. In specific, the client proxy code must convert the cantype data in the memory-object to a native representation structure that is used by the operation subroutines. For each operation, this conversion must be done before calling the operation subroutine. For write operations, the reverse translation must also be done to propagate the newly written object value.

Second, the parameter cantype conversion are transitory. That is, the converted input parameters are used only in the operation they are part of, and the cantype input parameters are not used after the conversion. The inverse is true for cantype conversion of output parameters in the message-passing approach. By contrast, the shared-memory approach would require conversion of the entire object data for every operation, and this converted object data would be used for each operation on that object. The larger the object is, the more efficient it would be to keep an in-memory copy of the native representation of the object, in parallel with the shared-memory cantype representation.

Keeping such a native-representation copy could reduce the number of wholesale conversions of the object data, but of course there would be a requirement for consistency with the cantype copy which is the actual data of the memory-object. Thus, the client proxy code would be doing conversions that are similar to what the manager is doing. That is, before the client proxy code calls an operation subroutine, there must be a check of the memory-object cantype data, to see if it has been changed (by another client) since the last operation, and if so to replace the in-memory native-representation copy with a freshly translated version of the cantype data.

6.5 Analysis of Approaches

As the preceding text shows, there is considerable and unavoidable overhead from translating to and from cantype data by both the manager and client proxy code. The lesson learned from examining client requirements is that an efficient distributed typed shared memory mechanism in Corbus is feasible among homogeneous hosts. On the other hand, heterogeneous distributed typed shared memory requires greater complexity of mechanism and decreased performance. There simply is no single mechanism for effective memory-mapped access to object data in a distributed heterogeneous environment. However, selective application of specific techniques can pro-

vide significant benefit in some cases.

6.5.1 Homogeneous Client/Manager Approach

To define what these cases are, it useful to recognize a further kind of heterogeneity: not all hosts in a Corbus system will be Corbus/Mach hosts. Corbus/Mach managers must still offer service to non-Mach Corbus clients. Therefore, even if a particular manager could effectively serve all Corbus/Mach clients in a memory-mapped mode of service, the manager would still have to offer an ordinary Corbus message-passing service to non-Mach clients. Therefore, one approach to heterogeneity is for memory-mapping object managers to offer memory-mapped services only to Corbus/Mach hosts on homogeneous hosts. Message-passing service would be provided to clients on heterogeneous Mach hosts and non-Mach hosts.

This approach essentially dispenses with the complex requirements and approaches to them that were presented in the previous section. This observation does not rule out cases where some of these approaches may provide enough benefit to justify the cost of heterogeneous distributed object cache management via paging. However, it is important to note that none of these complexities is strictly required for meaningful homogeneous use of paging for cache management.

Another important observation is to recognize the role of Corbus object replication. Clients of hosts of several different types can still benefit from memory-mapped service from a manager. Each different host type would have one manager on a host of that type offering memory-mapped service to clients on hosts of the same type. The various different managers would each be managing a replica of the same object, using the existing replication mechanisms of Corbus, and the approach described in Section 5.8 for combining paging and replication. The only new functionality required would be some awareness of each replica manager of the host type of other replica managers. This information would be used when the location mechanism resulted in a client on a host of type A contacting a manager of a host of type B. In these cases, the manager could forward the client to another replica manager on a host of type A, so that the client can get memory-mapped service.

A final note about replication and distributed memory-mapping is a comparison with replication and local memory-mapping discussed in Section 5.8. The primary difference is that client/manager IPC for replication processing is not local RPC but distributed RPC. This difference implies

some additional overhead. This increase in overhead is not likely to be significant, however, because the main bulk of the manager's replication processing involves distributed IPC with various replica managers on other host. One further instance of distributed IPC is not likely to increase the cost of replication beyond its benefit.

6.5.2 A Combined Approach

Another alternative strategy is to dispense entirely with distributed memory-mapping because of its difficulties with heterogeneity, and instead rely on some variant of the technique of client library management of caches as in Sprite, Photon, and NFS. However, doing so would unnecessarily forgo the benefits of distributed memory-mapping in homogeneous cases. A combined approach would utilize memory-mapping in homogeneous cases, and for heterogeneous cases would utilize client cache management rather than message-passing as in the previous section..

This approach would require some additional complexity in the manager and client proxy code, but the complexity would be of two separate mechanisms, each relatively simple and each used when appropriate. And two mechanisms are required in any case because of non-Mach Corbus hosts, as the previous section showed. (Note that non-Mach hosts can utilize the client caching approach because client caching does not rely on any Mach mechanisms.) Therefore, this combined approach simply substitutes client caching for messaging in heterogeneous cases of applications for which caching has already been identified as a benefit. For other applications where caching is not beneficial, the message-passing approach would still be used. Both the message-passing approach and the combined caching approach would be available as alternatives to application developers using the Corbus IDL tools.

Regardless of which approach to remote object data caching is chosen, Corbus can build an object data distribution infrastructure into client/server code.

Chapter 7

Real Time

7.1 Introduction

Modern real-time systems feature kernel-level thread management and adaptable scheduling functionality. A real-time system must complete a function at the proper time, accounting for computation time, resource access time, and other factors. Distributed system functionality is essential for real-time systems that require resource redundancy for fault tolerance. The Mach microkernel can meet these needs of real-time systems. It is also capable of distributed real-time functionality since it is extensible to a distributed environment in the manner required for Corbus/Mach.

Mach is capable of providing real-time system support, which falls into two areas, for distributed applications built on Corbus. First, a Mach system base can support real-time computation scheduling. Second, Mach thread management can provide scheduling coherence, a critical property of client/server real-time computing.

7.1.1 Real-time Scheduling

Real-time scheduling is supported in the Mach kernel by the use of a scheduling policy that schedules computation based on real-time properties such as deadline, priority, elapsed time, etc. However, different real-time environments often require different real-time scheduling policies. Therefore, the incorporation of a real-time policy is insufficient to demonstrate broad applicability to a range of real-time environments. As a result, scheduler replaceability has been a focus of real-time Mach research.

More recent research [5] demonstrates the insertion of a real-time scheduling policy into the Mach kernel, as a worked example that real-time scheduler replacement is feasible. Ongoing work at IBM [16] promises to merge scheduler replaceability with IBM's commercial microkernel development. As a result, a real-time Corbus/Mach system can be supported by a Mach microkernel that provides real-time scheduling.

7.1.2 Scheduling Coherence

Scheduling coherence is a distributed computation property in which all processing is scheduled in the same way, regardless of where the processing occurs. This is critical for real-time computation, lest separation of computational responsibilities come at the expense of uncoordinated execution. Priority inversion is an example of such a lack of coordination; it has been a significant factor in research of multi-threaded, client/server, real-time computation [17].

Scheduling coherence is particularly important in client/server computation, where real-time requirements are often asserted by client software. This client software may call several different servers to perform the computation. When clients wait for servers to complete a portion of some real-time computation, the resulting synchronous client/server communication is referred to a Remote Procedure Call (RPC). Hence, scheduling coherence and RPCs are closely related.

Recent Mach developments, especially thread migration, have led to Mach's ability to provide scheduling coherence. Therefore, Corbus/Mach can take advantage of this feature, together with real-time scheduling, to provide distributed object applications with real-time support.

Subsequent sections describe thread migration and several consequences of it for real-time-capable client/server software: coherent scheduling mechanisms, elimination of scheduling context switch for RPCs, new mechanisms for server threads, and data copying optimizations.

7.2 Thread Migration

Scheduling coherence is provided as a result of thread migration, a Mach IPC optimization approach pioneered at Carnegie Mellon University [22][23] as part of the decentralized real-time scheduling investigation, and subsequently refined at the University of Utah [9]. The Open Software Foundation (OSF) has implemented thread migration in the RPC feature of its

real-time Mach microkernel [11]. Commercial microkernel work at IBM has used thread migration as the essential Mach IPC mechanism. Any of these migrating-thread Mach kernels can be used as the basis for Corbus/Mach.

The remainder of this section is organized as follows. A detailed discussion of thread migration is followed by a description of the scheduling coherence consequences of thread migration. Finally, an example of migrating thread RPC that preserves scheduling coherence is provided.

7.2.1 A New Thread Model

There are two parts of a migrating Mach thread: an empty thread and a shuttle. A shuttle is the actual schedulable point of execution and an empty thread is the state of the thread pertinent to a specific task. When a shuttle is executing in a task, it has access to everything in the task's address space and it has some execution context within that address space. When a shuttle migrates from one task to another, that execution state is preserved, but remains inactive until the shuttle returns to the task. While the shuttle is executing in that other task, it has no special access to anything in the previous task.

An empty thread is the task-specific state of a single thread. An empty thread contains most of what is relevant about a thread at any given time; thus, the bulk of thread data are permanently bound to a task. The shuttle is the part of the thread that moves from empty thread to empty thread. Because the shuttle is just the scheduled point of execution, the only thread attributes that move between tasks are the scheduling attributes carried by the shuttle.

In order for a shuttle to migrate into a task, the task must have a vacant empty thread to which the shuttle can be bound. When a shuttle enters an empty thread, it carries no state from the previous task, other than the shuttle's scheduling attributes. The same is true when a shuttle exits an empty thread to return to the previous task. Also, once a shuttle has exited an empty thread, that empty thread retains no state of that shuttle. The information flow between the calling task and the callee task of a migrating-thread RPC is restricted to the thread's attributes, the calling task's RPC message, and the callee task's reply message. These two messages are exactly the same information that is passed in IPC without thread migration.

7.2.2 Scheduling Coherence As A Consequence of Thread Migration

The originally-intended benefit of the above RPC approach was to optimize RPC processing by the elimination of a context switch between the client and server threads. Scheduling coherence is a side-effect of the server's use of the clients shuttle, and hence the client's scheduling attributes. With scheduling coherence, the server executes in the same scheduling context as the client it is servicing, so no priority mismatches can occur. When the server thread completes its computation and returns from the RPC, the shuttle reverts to the client which then resumes execution– again, in the same scheduling context.

7.2.3 Client and Server RPC Processing

In the original Mach IPC model, executing threads would block on a kernel call to receive a message from a port. In server implementations, the following series of events were common: create a port (and obtain the receive right for it); make send rights to the port available to clients; create threads; and set each of them to execute the same procedure. First, the procedure would execute a blocking message receive on the port, wait until the message arrived, and then process the client's message and send a reply message. This sequence would be repeated when another message is received. A client, on the hand, sends a message to a server, and then does a blocking receive to wait for the server's reply message.

With the migrating threads approach, client and server processing is somewhat different. When a client uses migrating threads RPC to contact a server, the RPC is in a manner that appears to the client software as very similar to that described above: send a message, and wait for a reply. However, the kernel's migrating-thread RPC implementation moves the shuttle from the client's suspended thread to a thread in the server. In order to receive the RPC and the shuttle, the server software's port and thread processing is somewhat different than in the original model. For migrating threads, servers create *empty threads*. An empty thread has a dormant execution state and an undefined shuttle. When the server initialization software creates an empty thread, the empty thread is associated with a port and with some procedure to execute once an RPC on that port is made. When a client calls an RPC using a send right to a port, the kernel selects an empty thread bound to the receive right of the port. That empty

thread receives the shuttle and starts executing the previously associated procedure to process the RPC data.

In order to receive RPCs, *real-time* servers must allocate empty threads and bind them to both port receive rights and RPC-handling code. By doing so, servers can receive migrating- thread RPCs and perform computations for clients in a coherently scheduled manner. This is the only real-time-specific aspect of server processing that is general to object managers. In many cases, this is the *only* real-time aspect of server processing. In some cases, a server may place real-time constraints on its own processing, irrespective of clients' real-time requirements. A more likely case is a server that asserts real-time constraints when it is acting as a client to another server. However, such cases would be instances of application-specific real-time behavior of a service implementation, rather than generic server processing.

There is little impact on the existing Corbus code base as a result of this real-time requirement for object servers. Corbus object managers are built on a base of reusable code, called the manager skeleton, into which two structures are inserted. These are the IDL-generated communication software and empty procedure stubs, which developers fill with application-specific software. For thread migration and real-time, IDL-generated code would simply use the migrating threads IPC service. In some cases, this would not necessitate a change. For example, the IBM microkernel's IPC service is inherently coherently scheduled due to use of thread migration; therefore, no special IPC interfaces are needed.

7.3 Elimination of Scheduling Context Switch

The original intent of thread migration was to optimize synchronous message passing between clients and servers on Mach. Scheduling coherence was a beneficial result applicable to real-time systems. Thread migration also eliminates another part of the original Mach IPC model: the context switch between message sender and receiver. Since the context switch may have a negative effect on real-time computation, this is another benefit of thread migration.

A context switch occurs when one thread stops executing and the kernel initiates execution of another thread. The problem inherent with a context switch in RPC is that there is no guarantee that once the client thread suspends, the server thread will be the next to execute. Hence, given a set of real-time processing requirements, a scheduling context switch poses a

performance issue. One thread may be available to receive data but does not execute due to a lack of scheduling priority. The context switch could lead to undesirable and potentially unbounded delays in that processing.

Thread migration eliminates this potential delay between client message send and server message receive. The message send and receive occur in a single continuous thread of execution, in which the shuttle binds the client thread to the server thread. In addition, eliminating the scheduling context switch improves performance. It removes the kernel overhead associated with complete thread suspension and yielding to the scheduler in order to run a new thread.

7.4 Subsystem Registry

Subsystem registry is another real-time relevant optimization that results from thread migration. The focus of the optimization is the kernel's copying of RPC parameters. When the kernel copies RPC parameters from the client to the server and vice versa, it can copy only a data structure that is known to it. In Mach, the primary data structure is the Mach message. Hence, RPC parameters must be packed into a message on the client side via an activity known as marshalling. Likewise, on the server side, parameters can be used only after unmarshalling code unpacks the message into the original RPC parameters, each with its own type. Hence, data must be copied in three steps: the client copies from parameter form to message form; the kernel copies the message from the client to the server; the server copies from message form to parameter form. This process is repeated in reverse order upon the return of the RPC from the server to the client. A great deal of computational overhead is involved in marshalling and unmarshalling. This process is illustrated in Figure 7.1.

A "signature" may be defined as an RPC's list of parameters, including the type of each parameter. The signature provides information about the size and address of parameter data that must be moved by the kernel. A process may "register" a signature with the kernel so that the kernel can use the information to correctly copy parameters between RPC sender and receiver. Also, Mach signatures contain the location within parameter data of Mach port right values, which the kernel must translate as part of the copy. (Mach port rights are represented by integers which are mapped by the kernel to actual ports using a process-specific mapping similar to virtual memory mapping.)

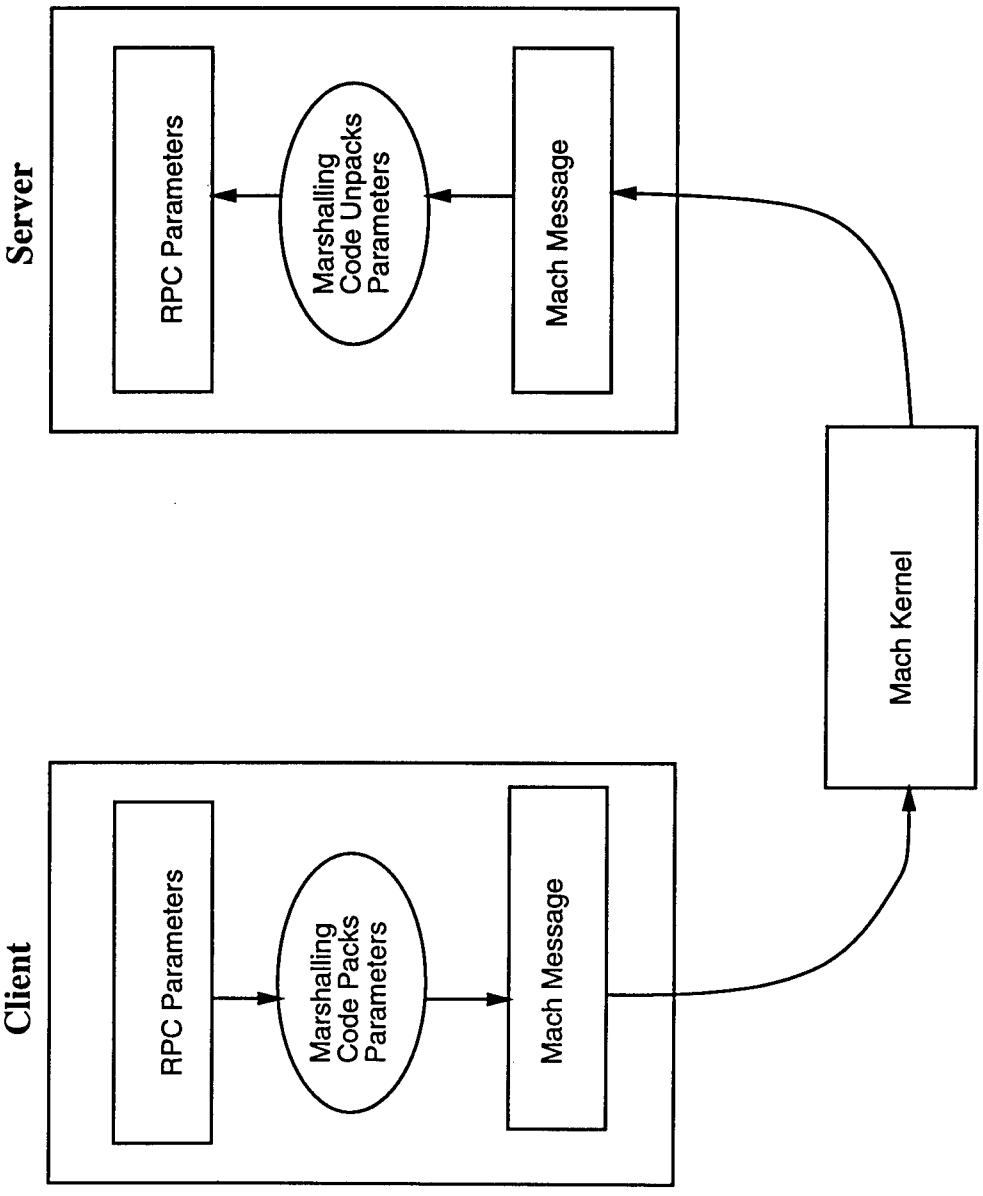

Figure 7.1: Dataflow of RPC Marshalling

A subsystem registry is used in an RPC interface to provide a presentation interface. It gives the kernel a data structure in which to copy parameters for a specific interface call from a client thread stack to a server thread stack. A subsystem registry may be defined as a data structure registered with the kernel by an RPC recipient. It ensures that the kernel has the information needed to copy RPC parameters and to call the appropriate RPC-handling functions of the RPC recipient.

This concept is closely related to the way in which an empty thread operates when it receives a message and how a thread is started. Each presentation interface has a signature associated with it. As stated above, this signature is expressed in the IDL; it consists of a parameter list. Since the server's initial RPC processing need not focus on unmarshalling and process determination for parameter data, it has a different focus. Server code can directly process a particular RPC; however, software must examine the RPC to determine which interface is involved and which server subroutine to call, and to branch to that subroutine. A subsystem registry contains the information needed for the kernel to perform this role. Hence, server code consists only of entries to RPC interface handlers. The kernel initiates server thread processing in the handler appropriate for each message.

In addition to the performance advantages of a subsystem registry, a reduction in development complexity is also gained. For a long time, IDL tools have taken an IDL interface definition and used the signature as the basis for the generation of marshalling and unmarshalling code. Given the ability to inform the kernel about signatures, marshalling code is unnecessary. Though a subsystem registry is application-dependent, a developer does not need to write code to deal with the interface. The developer could use an IDL generator to generate code to perform the subsystem registry functions.

Instead of marshalling code, IDL-generated code would provide the signature to the kernel when an RPC is called. The kernel would scan the signature of the RPC to locate the parameter block, which contains the definition of the signature. The parameter block is then copied onto the shuttle stack. Once a signature has been registered by the server, then subsequent copying of the signature data is unnecessary.

7.5 Original Mach Message Model vs. Migrating Thread Model

This section reviews the RPC processing changes that result from using thread migration techniques in the Mach kernel. In each step, the processing in the original message model is stated and contrasted with the consequences of thread migration.

1. Application code calls RPC function.

2. In the original mode, client-side RPC stub functions accept RPC parameters and marshalls them into the message structure. When subsystem registry is used in conjunction with migrating thread RPC, marshalling is not needed.

3. Client-side RPC function calls kernel interface to send message, or with subsystem registry send the RPC data as parameters rather than a message.

4. Kernel processing. With thread migration, the following kernel processing becomes unnecessary:

   (a) The kernel no longer needs to save message, and later copy the message data to the server task that is the recipient of the RPC. Instead the kernel copies RPC data without delay from the client to the server. With subsystem registry, the kernel can copy the RPC data in parameter form rather than as a message, this eliminating the need for marshalling.

   (b) The kernel no longer does context switch from sending thread to some other thread. Instead, the thread's shuttle remains as the current scheduling context, but is moved to the server task.

   (c) The kernel no longer needs to do a context switch to the server thread because the shuttle movement has already started the execution of server code.

5. In server-side RPC function, a server no longer needs to unmarshall and can go directly to application-specific code to process the RPC parameters, perform some computation, and derive the RPC return results.

6. The server finishes processing. Both the shuttle and the RPC return data flow back to the client in the same manner as steps (1) to (5).

7.6 Distributed RPC and Heterogeneity

The Corbus/Mach distributed IPC (dIPC) service will be based on an existing x-kernel implementation of networked Mach IPC (MachNetIPC). The dIPC service is the same as the microkernel's IPC; however, it provides for communication between tasks residing on different hosts in a distributed system.

The dIPC subsystem will be composed of separate micro-protocol modules bound together within the x-kernel protocol framework. Data moves through the graph in two passes. The first pass takes place on the sending host, where data moves from top to bottom. At the top, clients perform IPC. At the bottom, data traverses the network. The second pass occurs on the receiving host, where arriving data moves from bottom to top; network data is reassembled into IPC data which is sent to the receiver via local IPC.

The dIPC service will implement scheduling coherence by providing distributed emulation of local thread migration. This emulation is needed because the Mach microkernel is unaware of other hosts. Hence, a Mach thread cannot truly migrate to another host. However, the dIPC server can transport over the network the data that is required for a remote thread to execute just as if it were part of a migrating thread chain on another host.

When the dIPC server handles an IPC message that is the outgoing part of a migrating thread RPC, some specific actions are needed to maintain scheduling coherence. With each message's data that is transmitted to another host, the dIPC server must also send the sending thread's scheduling attributes. This information is used by the receiving dIPC server to perform distributed emulation of thread migration.

Once a thread migrates into the dIPC server and the outgoing message data is transmitted, the thread goes to sleep. On the receiving machine, the dIPC server uses a local thread to act for the sending thread on the other machine. This thread migrates from the dIPC server to the message's destination task. Before doing so, the dIPC server ensures that this thread has the same scheduling attributes as those that arrived with the message, i.e., the scheduling attributes of the sending thread. As a result, the receiving host's RPC processing is scheduled coherently with the computation in the sending thread on the other host.

It is important to keep in mind that many real-time systems have different scheduling requirements. As a result, one scheduler cannot support a wide range of real-time systems. Hence, scheduler replaceability (described in Section 1.1) is a critical Mach real-time feature. Because scheduling policy modules can be replaced at build time, Corbus can provide distributed application support for real-time software with a range of real-time requirements.

Due to Mach's flexibility with respect to real-time scheduling algorithms, an additional element of heterogeneity may arise in a distributed system. Not every host in a real-time distributed system may have the same real-time scheduling policy. Therefore, dIPC protocols must be able to detect when a host receives RPC data from another host that uses a different scheduling algorithm. This is necessary to prevent the receiving host from misinterpreting dIPC protocol data as scheduling parameter data and making use of spurious scheduling parameters. At a minimum, each host must maintain a list of other hosts known to have the same algorithm. More complex approaches include cognizance of multiple scheduling algorithms and attempts to perform partial mapping of parameters from one host's policy onto another host's policy. This latter approach attempts to approximate scheduling coherence.

7.7 Summary of Corbus Real-Time Factors

The following are recommendations for a Mach-based implementation of real-time requirements in a client-server environment such as Corbus.

- Employ a microkernel that readily supports real-time functionality. To this end, the microkernel must possess the following features:
  - Migrating threads. These are used to perform IPC.
  - Real-time scheduling policy. If such a policy does not exist currently, then the microkernel should be amenable to the addition of a new policy.
- Clients should use a migrating thread IPC service to implement RPCs with a server. This would be part of the client-side code generated via the IDL.

- The server should use the empty-thread functionality described in Section 2.1 to receive and process RPCs. This functionality would be part of server-side code generated from IDL.
- The system should use an x-kernel MachNetIPC implementation, which includes a protocol for the transport and use of scheduling parameters.
- The kernel should be able to identify network messages bearing RPCs with scheduling parameters from a heterogeneous host with a different scheduling policy.
- If subsystem registry techniques are employed, the IDL tools will be used to generate code for the subsystem registration mechanism, rather than for marshalling and unmarshalling.

If these recommendations can be implemented in a Mach base for Corbus, then real-time support can be almost transparent to application software, with the exception that some client software may need to assert timeliness requirements. Additional real-time work on object-oriented quality of service (described in Section 8) may eliminate this need for real-time awareness of distributed object software.

Chapter 8

Quality of Service for Objects

8.1 Introduction

Quality of Service for Objects (QuO) [18] [19] is an object-oriented resource management framework that BBN has recently applied to CORBA. QuO extends network-level Quality of Service(QoS) mechanisms by the addition of application-level mechanisms for resource management on a per-object basis. Because Corbus shares the CORBA orientation of QuO, Corbus is a natural candidate for the implementation of QuO mechanisms in a CORBA-compliant distributed application framework. Corbus's object-oriented nature and its application-level approach to distributed computing match the basic assumptions of QuO. Furthermore, Corbus/Mach offers additional benefits for an application platform with QuO mechanisms.

At the application level, QuO facilitates negotiations in support of three key benefits that are useful for Corbus applications — real-time operation, fault tolerance, and adaptive operation. In each of these cases, Corbus/Mach mechanisms offer significant enabling technology for each of these benefits.

The following sections provide background on QuO and Corbus, describe the Mach-specific aspects of the Corbus/Mach approach to QuO, and describe particular uses of QuO mechanisms that could be deployed in a Corbus/Mach system.

8.2 Background

The background for Corbus/Mach QuO is in three parts: Quality of Service in general, Quality of Service for Objects, and the way that the Corbus architecture can accommodate QuO.

8.2.1 Quality of Service

Quality of Service is a technique for system resource management in which various aspects of resource utilization are the basis for negotiations between resource providers and resource consumers. The purpose of implementing a QoS scheme is that a system's efficiency can be improved by tuning resource management to best accommodate the requirements of the resource consumers, given the capabilities of the resource providers. These requirements and capabilities are specified via "agreements" or "contracts" and address various aspects of resource utilization. By enforcing QoS contracts, "guarantees" regarding expected system behavior can be achieved. Although a guarantee might not be the most favorable for a system's or application's consumer, at the least it improves predictability about the application's or system's behavior.

The resources that might benefit from QoS management include services and system objects. Currently, communications researchers employ QoS schemes that address services at the socket level. For example throughput is an aspect of network service that might be specified in a socket-level QoS contract. In this case, bits per second would be an appropriate attribute for specifying and measuring service requirements.

The drawback of QoS is that it is too low-level. It does not address attributes that are important at the application level. This lack is a motivation for QuO.

8.2.2 Quality of Service for Objects

BBN's research into Quality of Service for CORBA objects [19] extends the above kind of QoS scheme so that it is suitable for the object-oriented design that is more and more frequently the basis for application programs. For example, whereas socket-level QoS might address usage characteristics such as network bandwidth, QuO can also address characteristics such as end-to-end delay, availability, and ordering.

Focusing on the application level has several important implications for

QuO. First, an obvious benefit of QuO is that application-level aspects of resource utilization may be specified in QuO contracts. For example, at the application level, resource utilization might be specified in object requests per second. Second, while network-level QoS is useful for performance attributes that are applicable to host-to-host connections, QuO is relevant to end-to-end communications. This is well-suited to RPC-type communications.

[19] describes a strategy for implementing QuO for the CORBA standard, although it does not describe implementation mechanisms in detail. For QuO, system-level usage patterns and requirements are specified by means of a description language (QDL) that is an extension of CORBA's Interface Description Language (IDL). QDL is used to specify *connection states* and two aspects of such states: performance guarantees, and QoS requirements.

For each state transition, QDL is also used to define a "handler" and to associate the handler with the transition. The binding occurs at connection time and then the handler associated with a transition will be called whenever the connection state changes. A QDL processor is used to build application-specific libraries that interact with ORB software to enforce the specifications. Such application-specific library code, sometimes called application proxy code, is more than a simple RPC stub library. Rather, QDL-derived proxy software includes additional functionality to call on ORB mechanisms that provide QoS functionality.

As a result of this approach, QuO support is implemented in a proxy software layer that lies above the layer of ORB software, but below the object request interface used by application client software. This layering preserves portability—a key goal for Corbus— by hiding behind the object request interface all the usage of the QuO mechanisms which may vary from system to system.

The enforcement of QuO agreements is implemented above the ORB layer by layers of smart proxies which are produced by processing the QuO agreement specifications. The QuO proxies are extensions to the ORB's proxies. QuO proxy software and ORB proxy software together facilitate communication with objects.

8.2.3 Corbus and QuO

The Corbus software architecture is structured in a way that can both accommodate the basic QuO approach, and utilize the advantages of the Cor-

bus/Mach communication architecture.

The most central Corbus ORB software consists of three primary components: library software that resides in server programs (the SCL), library software that resides in client programs (the CCL), and the Core which is separate software that handles message routing and object location. Application proxy software is layered on top of the CCL. Application client software calls object method invocation procedures. These procedures are implemented in proxy software which calls on the CCL to make object requests and receive replies from object managers.

Specification language tools are the means by which proxy software is created. Corbus uses the same CORBA IDL that the QuO work is based on. Therefore, it is feasible to extend Corbus's IDL tools to process QDL annotations in IDL specifications in order to generate application-specific QuO software for application proxies. This QuO software would be in addition to the RPC stub functionality of application proxies that are currently generated by the Corbus IDL tools.

A convenient way to incorporate QDL connection contract annotations in IDL specifications is by embedding them within comments. QDL annotations require only a very simple syntax to specify the connection states, handlers, and QoS attributes. For example:

```
\\ Actual comment text
\\ BEGIN QDL extensions to IDL
\\ LowDelayConnection1 <- ( ObjectA,
\\       Attributes( Priority = 99;
\\                   Delay = 50;
\\                   MethodsPerSecond = 100)
\\       Handlers( State1-to-State2 = Handler1;
\\                 State2-to-State3 = Handler2) );
\\ END QDL extensions to IDL
\\ BEGIN other extensions to IDL
\\ ...
\\ END other extensions to IDL
\\ Text ending the CORBA IDL comment block
```

The Core is the part of Corbus that actually calls on the O/S and networking services to move client/server data. Client and server processes themselves communicate only with the local Core, using the functionality of the CCL or SCL to do so. Therefore, the Core is the part of Corbus

that is suitable for the functionality of using QoS features to enforce QuO agreements.

8.3 Corbus/Mach Approach to QuO

The separation between the Core and the application proxy is a separation between the software that is aware of QuO agreements (the proxy) and the software that is capable of using QoS mechanisms (the Core). Because of this architectural separation, there are two different ways that QoS mechanisms can be used in Corbus/Mach for QuO.

8.3.1 Two Approaches

In both approaches, application proxy software would make QoS-aware use of communication services, as in the basic QuO architecture. Proxy software would implement object method invocations by packing the various parameters into an object request message, and sending that message via call to some communication service. In addition, the proxy's use of the communication service would include the specification of object-specific QoS information. The difference between the two approaches is the difference between the communication service called on by the proxy software.

In the first approach, the Corbus CCL is used as in standard Corbus, as a means of forwarding object requests to the Core, which uses the actual system communication service. Therefore, object-specific QoS data would have to send to the Core by the CCL in addition to object request messages. The Core would perform all QoS usage in support of QuO requirements, based on information forwarded to it by the CCL.

The second approach would be an extension of the overall approach described in Section 3, in which the CCL would make direct use of O/S and network services for communication, eliminating (in many cases) the Core as an intermediate party. As a result, clients and servers can take responsibility for their own QuO agreements, rather than relying on the Core. When communicating with servers, a client's CCL can send the object request message itself. The CCL would directly use communication services and use object-specific QoS data to enforce QuO agreements.

Because of the centrality of the Core in the Corbus architecture, the Core must have a role in maintaining and enforcing QuO agreements. The second approach simply moves part of this role from the Core to the CCL in cases where this is expedient. Such expedient cases are those where a

client's CCL already has the ability to communicate directly with an object's managers, as a result of the Core's handling of a previous request on that object. Therefore, the distinction between the two approaches is that the first approach is used when a client makes its first request on an object, while the second approach is used for subsequent requests.

In the approach of Section 3, the Core is not a pure location broker, but also acts as a message switch for object requests for which object location is required. As a result, the Core is involved in the initial communication between client and object manager. Therefore, the Core is involved in the definition of QuO contracts, and maintenance of QuO agreements for initial object requests. In a slightly alternative approach, the Core would not act in this QuO role if it were to act purely as a location broker, i.e. informing the client where the manager is, and letting the client make the initial communication.

In addition, the first approach must be used in cases where the object's manager is on a non-Mach Corbus host. In these cases, the object manager is not able to receive direct communication with the client, and therefore the Core must act in its intermediary role. In other words, the Core must act as a message switch (and not merely as a location broker) in heterogeneous cases. In addition, the Core may have to deal with the possibility that non-Mach Corbus hosts may not support QuO in a compatible manner because the underlying O/S does not provide the same QoS mechanisms that are available on Corbus/Mach hosts. If this is the case, then it would be necessary for a Corbus/Mach host's QuO specification to note those hosts with which there can be no QoS agreements.

Additional sophistication would be needed to support partial QoS agreements between Corbus/Mach hosts and non-Mach Corbus hosts that support a subset of the desired QoS agreement. In this case, a message-switching Core would need to note, for each connection, the expected level of support for QoS. If the IDL extensions to support QuO are embedded within standard IDL comments, as described above, then it would be a simple matter to include notations to support interoperability among heterogeneous hosts, including non-QoS hosts.

Negotiation in support of communication among distributed Corbus/Mach hosts could be implemented by an x-kernel implementation of MachNetIPC. Negotiation is discussed further in Section 8.3.3.

8.3.2 QoS Mechanisms on Corbus/Mach

The QoS mechanisms upon which QuO is based are largely implemented in communication protocols. For Corbus/Mach hosts, the *x*-kernel provides an excellent place for protocol software to execute. A key feature of the *x*-kernel is that it is designed for extensibility and for adding new functionality without changing existing protocol implementations. Because QoS mechanisms can interact with existing communication protocols, the *x*-kernel's modularity and protocol composability are critical to ease of deployment of QoS. Because of the *x*-kernel's extensibility, support for QoS agreements could be added with minimal interference with existing protocol implementations. QoS support on Corbus/Mach can also be straightforwardly integrated with the basic communication service between Mach hosts: the *x*-kernel implementation of MachNetIPC which was designed for transport of RPCs and, therefore, shares QuO's focus on end-to-end communication.

For communication between Corbus/Mach and non-Mach Corbus hosts, the *x*-kernel's framework will likely simplify the modification of the existing communication protocols in order to support QoS. As noted above, this support will vary from connection to connection, depending on each non-Mach Corbus host's level of support for QoS attributes.

8.3.3 Negotiation of QoS

Several types of negotiation can be envisioned for Corbus/Mach. This subsection explores the possibility of QoS negotiation and identifies some potential uses.

The purpose of QoS negotiation is to achieve optimal connections by trading off QoS attributes in response to existing constraints. Connections may be optimal only from the point-of-view of an object designer or they may be optimal from the points-of-view of both object clients and object servers, subject to any relevant external constraints. Thus an "optimal" connection might actually provide fairly poor QoS, but if the only alternative is no connection, then poor QoS is optimal.

We use the term, "negotiation," to describe a range of decision-making. Ideally, the range of potential negotiation should extend to balancing the changing needs and capabilities of both clients and servers. This would require an enhancement to QuO in which clients' also levy requirements, as do servers. However, useful negotiation can also be achieved using the one-sided agreements of QuO, in which only constraints on clients are specified,

rather than also specifying clients' requirements.

In support of QoS negotiation, several object connections may be defined and opened. Each connection might provide a different blend of QoS, strong in some aspects but weak in others. Several connections might be viable at a single point in time. For example, there could be several connections that are all capable of guaranteeing the desired level of throughput for a given client. For many clients, that level of service might be sufficiently "optimal" that no further negotiation is warranted. For other clients, it might be important enough to achieve a higher level of some other QoS attribute that a different connection, providing an alternative combination of levels of QoS, is worth negotiating for.

Handlers may be invoked in response to the changing state of a connection. In order to support QoS negotiation, the role of the handlers could be expanded. It is the handlers that will accomplish negotiations by initiating connections with the desired combinations of QoS attributes. Rather than responding only to invocations that result from changes in system state, handlers can initiate connection upgrades as more optimal connections become viable. In the case of two-sided negotiation, one way for a client to acquire knowledge about the viability of connections would be via polling, at some interval. Alternatively, servers could "advertise," via a broadcast mechanism, the combinations of QoS that could be provided.

The flexibility of the x-kernel protocol framework supports richness in the nature of negotiation of QoS attributes among clients and servers. As described in Section 4.1, at different times different parts of a protocol graph may be utilized. It was explained that this provides support for varying strengths of transport protocols. By the same token, this flexibility can support varying strengths of QoS. For communication with a host about whose QoS much is known, more complex negotiations can take place. For communication with lesser-known hosts, simpler negotiations are warranted.

8.4 QuO Features for Corbus/Mach

Because Corbus is a CORBA-compliant ORB, and because QuO has already been defined for CORBA, there are potentially several ways in which QuO can be beneficially used in Corbus. This section describes the benefits of QuO usages in three contexts: real-time operation, fault tolerance, and adaptive behavior. Different aspects of QuO support each of these benefits. For real-time operation, the ability of QuO to incorporate and augment QoS

attributes is beneficial. For fault tolerance, it is the state transition handlers that provide the benefit. For adaptive operation, the benefit is that the status of the QoS attributes is made visible at the application layer, so the support comes from both of the above aspects.

8.4.1 QuO for Real-Time Operation in Corbus Applications

Real-time operation is a significant factor in achieving interoperability among heterogeneous systems. Process scheduling becomes especially difficult because different real-time systems are likely to have different scheduling requirements.[1] Real-time scheduling may be based on attributes that don't exist for non-real-time schedulers. Likewise, the scheduling attributes of one real-time system may not be present in another system. Therefore, there is a significant issue not only for propagating scheduling requirements between client host and object manager host, but also for dealing with variations in real-time behavior of different systems.

For Corbus, QuO resource management provides a simple mechanism to incorporate additional scheduling attributes in support of real-time operation. For example, with a priority-based scheduler, a priority attribute would be added to the system's other QoS attributes. Thus, a client's usage pattern would need to include a description of priority requirements in addition to other expected scheduling information. Processing the current status of the deadline could require the modification of existing scheduler mechanisms, but QuO also provides hooks for such processing. A different handler could be called as a connection's deadline interval approaches a critical threshold.

8.4.2 QuO for Fault Tolerance in Corbus Applications

Although partial failures and complete failures are facts of life in most distributed applications today, the ability of applications to function in the face of failure is not impressive. What is needed is a method for describing and quantifying expectations about service and resource provision and usage.

[1] Several areas of O/S research, including the Mach-based Triad system [5], are addressing the use of a replaceable scheduling framework. Such a framework within the Mach kernel would separate the actual scheduling mechanisms from the algorithm that determines order of execution. The algorithm software interacts with the actual scheduling mechanism via a well-defined modular interface. As a result, the real-time requirements of various systems can be met by the insertion into the framework of the particular algorithm required for a system.

QuO lends itself readily to describing performance attributes that affect a system's ability to function when the services and resources it relies on fail to some extent. But the main benefit of QuO for fault tolerance is that, by means of handlers and handler generators, QuO improves a system's ability to function under adverse conditions.

The QuO mechanism that supports fault tolerance in the ability to associate handlers with connection status transition specifications. The handlers are invoked when the associated transition occurs.

The object orientation of QuO might also be considered a "mechanism" that improves fault tolerant operation. Object orientation enables QuO to abandon the monolithic QoS management schemes of (non-object-oriented) client-server systems. With QuO, each object manages its own QuO attributes, in a sense. Server designers, who know the capabilities, weaknesses, and requirements of their server and its objects, can best design handlers to cope with failures.

With respect to fault tolerance, the main distinction between the local and distributed cases is that the distributed case introduces vastly many more possibilities for partial failures. This enhances the value of QuO's ability to address fault tolerance. With respect to implementation, distributed objects by means of proxies. If an object's proxy can provide locally QoS information, then the local client has an advantage in coping with failures in the remote version of the object.

8.4.3 QuO for Adaptive Behavior in Corbus Applications

Adaptive behavior in the operational phase of a system's life cycle is desirable for several reasons, all of which derive from the necessity and difficulty of predicting conditions and circumstances that might arise during that phase. Systems need to be designed so that they can perform to the required extent given 1) predicted deviations from the most desirable operating conditions, 2) unexpected, but known, anomalies, and 3) unknown adverse conditions. All of these can negatively affect the availability of resources and the performance of servers. Assuming that some variance in these is tolerable (at least in some operational modes), the ability of a system to adapt to variations that result from these three circumstances improves its ability to perform under these circumstances and can help ensure optimum operating efficiency.

The need for system adaptivity is one of the key motivations of BBN's research into QuO. Many distributed applications operate in an environment that is dynamic. Even CORBA's interoperability does not provide sufficient

support in this regard. As an example, it is noted that without the advent of QuO, most non-multimedia-based distributed applications required complex and tedious hand-tuning in order to handle variations in resource performance and availability.

For Corbus, adaptive behavior is a desirable characteristic in part because it furthers the goal of interoperability. If a Corbus application can assimilate knowledge about variances in performance among its distributed hosts, then the degradation of its overall performance in the face of such variances may be reduced (or at least reduceable).

QuO supports this ability of systems to adapt by providing a means of specifying multiple or multi-state patterns of resource usage and resource provision by clients and servers.

The QuO mechanism that can be used to provide support for system adaptivity is the QoS Description Language's (QDL's) system state specifications. These specifications describe the status of a client's connection (possibly one of many) to an object with respect to the attributes covered by the QoS agreement. To enable applications to adapt to changes in the operating environment, the actual status of QoS attributes is made available by QuO at the application level. This enables an application to cope with variances in resource availability by altering its behavior in pre-defined ways.

Although adaptive behavior is generally considered with respect to distributed systems in which each host must cope with variability in all the others, QuO can easily support adaptive behavior among co-located clients and servers. Adaptivity in local communication is achieved by means of handlers that may be associated with transitions between states. These handlers were described in Section 8.2.2.

The key advantage of QuO for adaptive behavior is the ability to take advantage of knowledge about **external** usage characteristics, i.e. characteristics of another process, application, or host. In the case of communication between Corbus/Mach and non-Mach hosts, one can assume that the non-Mach hosts are hosts whose performance has not been quantified in terms of a QuO contract. In this case, the benefit is scant, but the use of QuO at least offers an application the ability to specify its **lack** of knowledge. In some cases, it is likely that even this small increase in knowledge will yield some benefit in interoperation.

Chapter 9

Distributed Object Security

9.1 Introduction

Many distributed systems have a requirement to provide controlled access to a defined set of resources, e.g., finance and personnel data, in accordance with a specific security policy. At the same time, budgetary and system management constraints may limit the amount of hardware that can be purchased and managed, thus eliminating separate systems as a solution. Depending on user and system requirements, it may also be desirable to support automated, controlled sharing of certain types of data between systems or separate user groups.

These basic security requirements can be met by a secure access control scheme implemented via the establishment of separate domains of execution. Definition of separate domains can be accomplished using Mach microkernel mechanisms that specifically support domain separation. These mechanisms can be used in a Corbus/Mach system without interfering with Corbus functionality. In addition, these Mach mechanisms can provide different operating system (O/S) personalities within each domain. As a result, Corbus-based application software can be deployed on its required O/S, while users within a domain can use the O/S interface with which they are most familiar.

This section provides an outline of the Mach approach to domain separation, and illustrates how it can operate without impact on Corbus mechanisms.

9.2 Security Identifiers

The security identifier (or secID) is the Mach mechanism that can be used to establish separate domains of execution among distributed systems and applications. This mechanism is a simple ID tag that has been implemented in the IBM microkernel, the OSF Mach++ kernel [21], and recently applied [5] to the OSF real-time kernel [11]. The Mach kernel assigns also, spell out IPC and RPC? each task a secID upon creation, and tags all IPC (whether messages or RPCs) with the secID of the sending task. The Mach kernel does not interpret secIDs or base any access control decisions on them; rather, it ensures secID integrity and makes secIDs available to higher-level components such as servers.

Use of secIDs for high-level components falls into two basic categories. The first category is definition of secIDs. When a task is created, the Mach kernel assigns the new task its secID in either one of two ways. The default way is for the secID to be inherited from the task which requested the new task's creation. The second way involves the use of a kernel privilege which allows a task holding the privilege to dictate the secID of tasks it creates. Granting of this privilege to tasks is a function of the system initialization mechanism. Subsequent use of the privilege is dependent on the functionality of the privileged tasks. Normally, a task's secID will be based on some security-relevant attributes of the task, and the mapping between the secID and those attributes will be maintained by some component related to the component or components which define secIDs.

The second category of secID usage is message tags. Receivers of Mach IPC can examine the message or RPC for the sender's secID that the kernel tagged into the IPC. This secID can be interpreted (possibly by recourse to some service mapping secIDs to security-relevant attributes), and the interpretations used by the receiver to determine the processing related to the IPC. For example, access control components can base access control decisions on secID information.

Distributed secIDs message tagging is performed by the MachNetIPC Server, described in Section 4. In addition to its transfer of message/RPC data and port rights, the MachNetIPC Server also transports the IPC sender secID from sender to receiver. First, when the MachNetIPC Server receives IPC from a local task, it extracts that task's secID. Second, when the MachNetIPC Server sends the IPC data over the network, it includes that secID. Third, the receiving MachNetIPC Server extracts the secID from the network message containing the IPC. Fourth, when the MachNetIPC Server

delivers the IPC to a local task, it tags the local IPC with the extracted secID. To do this tagging, the MachNetIPC Server uses a kernel privilege to override the default inheritance of a message's secID from the sending task. This privilege is needed only by the MachNetIPC Server.

9.3 Domain Definition

The concept of domain definition is presented in the following paragraphs in terms of both a single host environment and a distributed environment. Multiple domains per person and controlled sharing between domains are also discussed.

9.3.1 Single Host Environment

Figure 9.1 illustrates a simple type of domain definition, i.e., one host with two separate domains. In this example, there is neither distributed nor security ID functionality; the distributed case is described below. Each domain consists of its own O/S personality, Corbus clients and object managers, and a Corbus Core. Each domain's Core sets up clients and managers to communicate within that domain. The two Cores (one in each domain) are unaware of each other's existence, and no modifications to the software are needed.

There is no inter-domain communication. Separation between the domains is ensured by the fact that capabilities for communication between the domains, i.e., port rights, are not granted by the Core. During system initialization, the Mach system initialization task creates each domain as an O/S personality (or personalities), a Corbus Core, and one or more Corbus managers. Within each domain, the Core and managers can communicate with one another, and the O/S personality(s) given a capability for communication with the Core. Clients are created by an O/S personality, after a user starts a session on a display device managed by the personality. Client tasks can only communicate with the following tasks: personality task(s); any tasks that the personality task(s) can communicate with, i.e. the Core; and any further tasks that the Core can communicate with, i.e. object managers. Because no task in one domain is given a port right for a task in another domain, further tasks created in a domain are similarly isolated from other domains. If a NetName server is used (as described in Section 3) in Corbus/Mach, there would be a separate server in each domain. Because each NetName Server is part of a domain (and isolated within it just as the

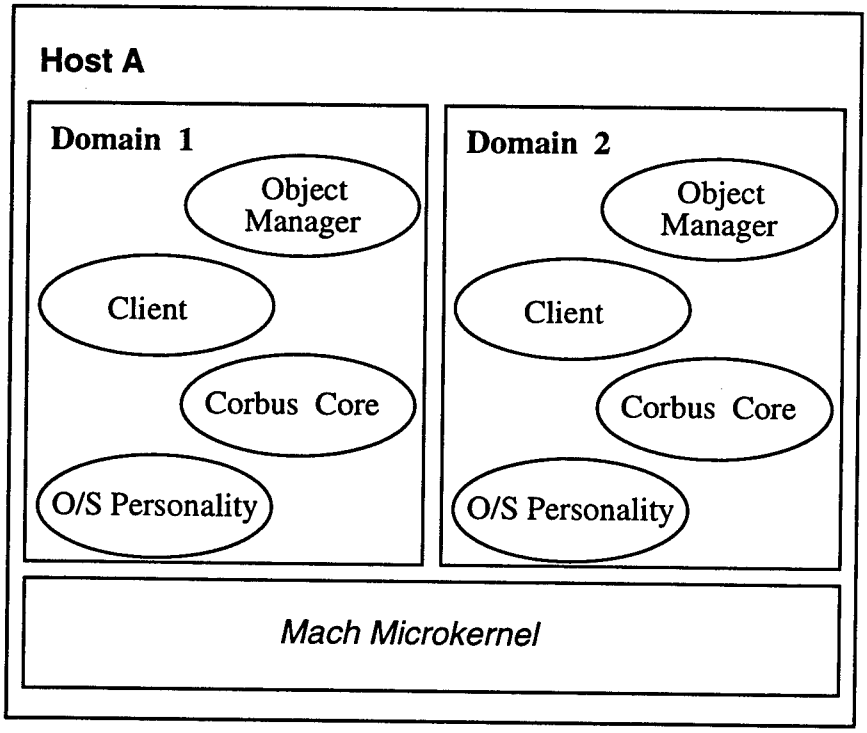

Figure 9.1: Separate Domains on One Host

other tasks are), the name service cannot be used to establish communication between domains.

9.3.2 Distributed Environment

A distributed environment would be one in which there are several hosts similar to that shown in Figure 9.1, but with two changes. The first change is the addition of the Mach NetIPC Server, which would provide Mach IPC between these hosts. There would be one NetIPC Server per host, with the NetIPC Server placed in the lower architectural layer with the Mach kernel. The second change is the role of the NetName Server. As mentioned in Section 4, this service (or one similar to it) should be used to enable the Corbus Core on each host to to be contacted by the Cores of other hosts. This initial contact is the initial step that enables forwarding of object requests between hosts. A secID-aware NetName Server, in combination with the secID functionality of the NetIPC Server, would enforce domain separation on client/server interactions between hosts. The secID-aware NetName Server, in order to serve multiple domains and enforce separation between them, is placed in the lower architectural layer (with the Mach kernel and NetIPC server) as illustrated in Figure 9.2, rather than being instantiated in every domain as in Figure 9.1.

In a distributed environment, each domain would be given a separate secID, and each task in each domain would be created with the secID of the domain it is in. For tasks created during system initialization, startup software would assign secIDs using the secID privilege described above. None of the tasks in a domain would be given this privilege, so that every one created in the domain later would inherit the secID of the domain.

During system initialization, the Corbus Core makes NetName requests to obtain ports to Cores on other hosts. The following series of steps illustrates the secID-relevant functionality of this initialization. The notation ${Core^1}_A$ denotes the Core in domain 1 on Host A. Similarly, $NetName_A$ and $NetIPC_A$ refer to the NetName and NetIPC Servers on Host A. These lack a domain superscript notation because they are not in a domain but rather underly all of them as shown in Figure 9.2.

1. In each domain on each host, the Core performs a NetName registry under the name "CorbusCore." Note that the NetName Server allows multiple entities to register under the same name on the same host as long as these entities have different security IDs.

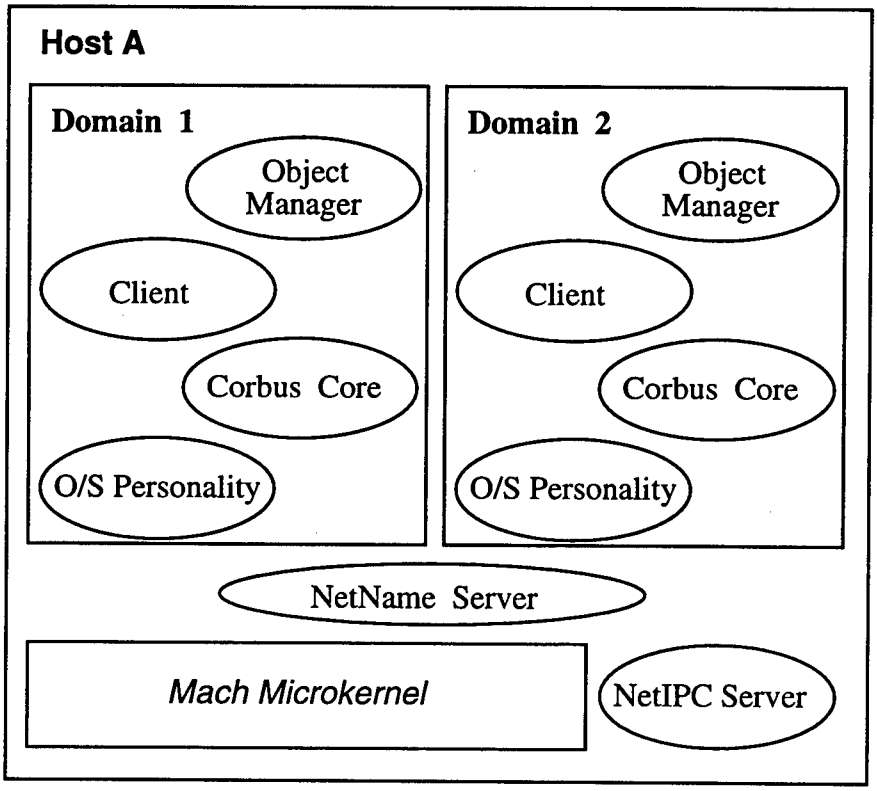

Figure 9.2: Separate Domains on One Host

2. $Core^1{}_A$ requests a NetName lookup for the name "CorbusCore" and the host B.

3. $NetName_A$ forwards the lookup request to $NetName_B$

4. $NetIPC_A$ transports the forwarded request to $NetIPC_B$, including the secID for Domain 1.

5. $NetIPC_B$ delivers the forwarded lookup request to $NetName_B$, including setting the sender secID tag to the secID for Domain 1.

6. $NetName_B$ examines the name in the request (specified by $Core^1{}_A$) and the secID of the request (added by $NetIPC_A$) and finds a port for $Core^1{}_B$.

7. A right to this port is returned by $NetName_B$ in a message that goes from $NetName_B$ to $NetIPC_B$ to $NetIPC_A$ to $NetName_A$ to $Core^1{}_A$, which receives the port right and can subsequently communicate with $Core^1{}_B$.

As a result of this series of steps, a Core can establish distributed communication with Cores on other hosts, but only in cases where the Cores have the same secID. This restriction is enforced by the secID-aware NetName Server, and supported by the NetIPC Server's secID tagging feature.

Note that other host configurations are possible as well. For a user that has access to only one domain, a workstation may well be configured with only that one domain. The user can execute client software that communicates with object managers on other hosts that have the same domain as the client. The only security-related software on such hosts would be the code in the NetIPC server that tags every outgoing network message with the secID of the single domain of the workstation.

9.3.3 Multiple Domains Per Person

The architectural views described above can be modified so that individual people can operate in multiple domains, though not simultaneously. On hosts like those illustrated in the above figures, it is possible for one person to be able to operate in multiple domains, if there are multiple domains in which a person can start a session with a personality. However, because the domains are strictly separated, each display device (e.g. terminal, console, networked virtual terminal) is allocated to exactly one domain (and perhaps

to one personality within it). Therefore, for a user to actually use access to multiple domains, the user must use multiple physically distinct displays. This section describes how this restriction can be overcome so that one user workstation could be used to access multiple domains from the same workstation console device. A practical example of this would be a person needing to access both the finance domain and the general corporate domain in which employees communicate via email.

This multi-domain capability would require the addition of a small new security-aware component that manages multiple sessions. The responsibility of this new Session Manager component would be the control of the console display device.[1] The Session Manager would allow dynamic enforcement of separation, in contrast to the static form of separation established during system initialization when each display is allocated to one domain for the life of system. Instead, the session would ensure that the console display is accessible to one domain *at a time*. In other words, the user can use the console display to have a session in one domain, and then switch to a session on another domain with the same display device. This allows users to operate in multiple domains from one desktop, rather than having a separate monitor/keyboard/mouse allocated statically for each separate domains the user can operate in.

Requirements for this component would include access from the Mach microkernel to the display device(s) and the ability to pass access to the device(s) to O/S personality software. In addition, the Session Manager would require a means by which the user (or the personality software acting on behalf of the user) can indicate that the current session is done, and the Session Manager should resume control of the display in order to query the user about which domain to operate in for the next session. Given these requirements, the Session Manager would be able to operate outside O/S personalities and domains in order to time-multiplex access by the domains to the console device.

Impact on the other components would be as follows. The NetName and NetIPC Servers would not be effected at all. The system initialization software would have to start up the Session Manager, give it the ability to control the console, and give some communication capability to either be signaled by personalities, or to signal them. Personality software may be

[1]The Session Manager could also control multiple display devices, for hosts which have several terminals. For purposes of explanation, however, we assume that the host in question is a workstation with only one display device.

effected, in that it might need to yield access of the display device to the Session Manager. However, personality software need not be effected, if the Session Manager can control the execution of personality software. That is, the user could signal the Session Manager (perhaps via special keystroke sequence recognized by the display device driver), and the Session Manager would suspend execution of the current session, and start a new session in a new domain.

9.3.4 Controlled Sharing Between Domains

Controlled data sharing between domains would occur when a client in one domain performs an object request for an object managed by server in another domain. In order to control these interactions, there must be a new component that permits or denies requests based on which domain the request is from and which domain the request is to. This component, called the Domain Object Request Switch (DORS) for the current discussion, would enforce a policy which describes all of the allowed cases of one domain being able to make a request on an object of another domain.

The primary interface of the DORS would be with the Cores of the various domains on a host, as illustrated in Figure 9.3. A Core in one domain would be able to forward to the DORS an object request for an object in another domain. The DORS would accept these requests, determine whether the request is permissible, and if so would send the request to the Core of the target domain. If there were no instance of the target domain on the local host, the DORS could use the NetName service to find a Core of the requisite domain on another host. This in turn requires that the secID of the DORS be some distinguished value that the NetName Server will recognize and allow lookup to any domain.

A similar control flow in the reverse direction would occur for replies to object requests. Each reply would be tied to a particular request, and the DORS would forward the reply from the object manager's domain's Core to the client's domain's Core. Additional interaction between the DORS and the target domain Core may be required to ensure that the object manager is the only entity in the target domain that is able to send a reply back into the original domain. For example, the restricted port rights of the IBM microkernel (port rights that can only be transferred once) could be used to enforce this restriction.

Aside from the existence of the DORS, the other significant change for controlled sharing is a change to the Core itself. In all other functionality

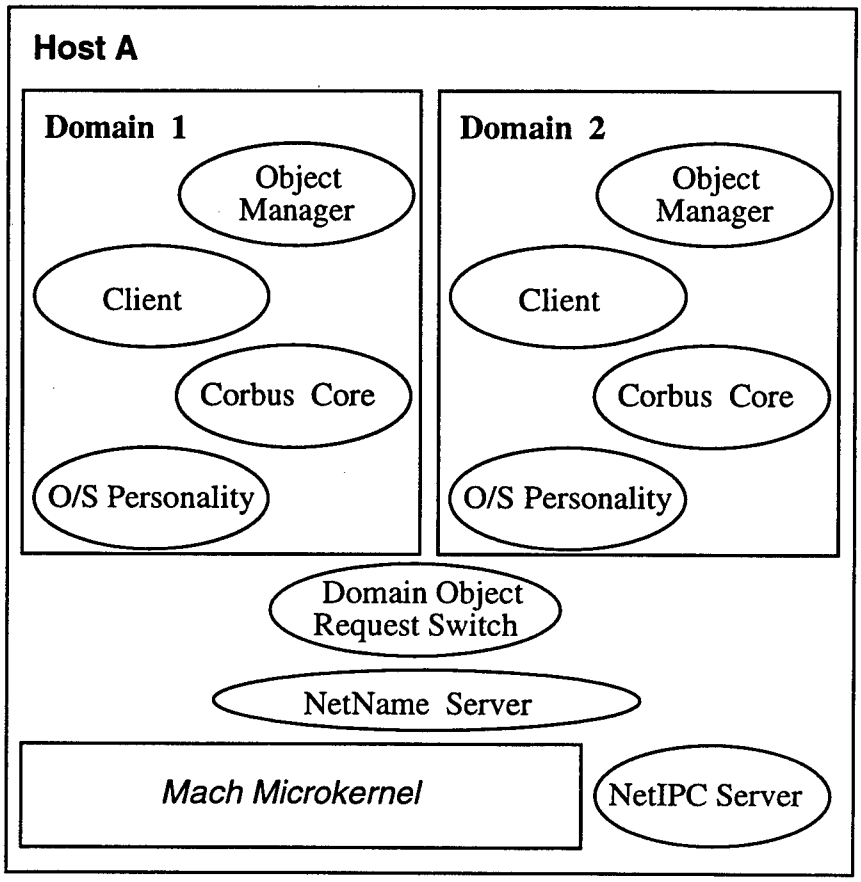

Figure 9.3: Separate Domains on One Host

described in this chapter, no Corbus software needed to be changed for Corbus to function in a multi-domain environment. In this case, however, the Core is required to do two new actions: first, use a new mechanism to identify object requests that are for another domain; second, use the interface with the DORS to send such requests and to receive replies to them. A variety of approaches could be used for identification of the target domain of a request. Perhaps the simplest is for object handles to include an indication of the object's domain.

Bibliography

[1] N. C. Hutchinson, L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering, 17(1)*, pp. 64-76, Jan. 1991.

[2] H. Orman, E. Menze III, S. O'Malley, L. Peterson, "A Fast and General Implementation of Mach IPC in a Network," *Proceedings of the USENIX Mach III Symposium*, pp. 75-88, Apr. 1993.

[3] J.M. Stevenson, D.P. Julin, "Mach-US: Unix on Generic OS Object Servers," *Proceedings of the USENIX Technical Conference on Unix and Advanced Computing Systems*, pp. 119-130, Jan. 1995.

[4] Trusted Information Systems, Inc., "Trusted Mach System Architecture," Technical Report TIS TMACH Edoc-0001-93B, Trusted Information Systems, Inc., Glenwood, MD, May 1993.

[5] E.J. Sebes, T. Benzel, *et. al.*, "The Triad System: the Design of a Distributed, Real-Time, Trusted System," Proceedings of the Aerospace Computer Security Applications Conference, December 1995.

[6] Michael Bushnell, "Towards a New Strategy of Operating System Design," Free Software Foundation, Inc., 1994.

[7] Johannes Helander, "Unix under Mach: The LITES Server," Master's Thesis, Helsinki University of Technology Department of Computer Science. (Available from `http://www.cs.hut.fi/~jvh/lites.MASTERS.ps`.)

[8] Jay Lepreau, Mike Hibler, Bryan Ford, Jeffrey Law, "In-Kernel Servers on Mach 3.0: Implementation and Performance," Mach III Symposium Proceedings, April 19-23, 1993, Santa Fe, New Mexico.

[9] Bryan Ford, Jay Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model," Proceedings of USENIX Mach Symposium, January 1994.

[10] Michaeal Condict, David Mitchell, Franklin Reynolds, "Optimizing Performance of Mach-based Systems by Server Co-Location: a Detailed Design," OSF Research Institute Operating Systems Collected Papers Vol. 2, October 1993.

[11] E. Burke, M. Condict, D. Mitchell, F. Reynolds, P. Watkins, W. Wilcox, "RPC Design for Real-Time Mach (Draft)," OSF Research Institute Operating Systems Collected Papers Vol. 3, April 1994.

[12] H. Orman, E. Menze III, S. O'Malley, L. Peterson, "A Fast and General Implementation of Mach IPC in a Network" Mach III Symposium Proceedings, April 19-23, 1993, Santa Fe, New Mexico.

[13] R. Sansome, D. Julin, R. Rashid, "Extending a Capability Based System into a Network Environment," Carnegie-Mellon University, 24 April 1986.

[14] D. Reed, R. Kanodia. "Synchronization With Eventcounts and Sequencers," *Communications of the ACM*, February 1979.

[15] G. Benson, "An Optimal Solution to the Secure Read-Writer Problem," Proceedings of IEEE Symposium on Research in Security and Privacy, May 1992.

[16] C.J. Paul, "IBM Microkernel for Scalable Real-Time Distributed Systems," Rome Laboratories Technology Exachnge Meeting, 29 Novemeber 1995.

[17] H. Tokuda, T. Nakajima, "Evaluation of Real-Time Synchronization in Real-Time Mach," Proceedings of USENIX Mach Symposium, November 1991.

[18] J.A. Zinky, D.E. Bakken, R. Schantz, "Overview of Quality of Service for Distributed Objects," 1995 Dual Use Technologies Conference, Utica, N.Y., May 23, 1995

[19] J.A. Zinky, D.E. Bakken, R. Schantz, "Supporting Quality of Service for CORBA Objects," BBN Systems and Technologies Report No. 8119.

[20] Floyd, Richard et. al., "Future Directions for Replication in Cronus," BBN Systems and Technologies Corporation, 16 April 1990.

[21] Open Software Foundation, *Mach Kernel High Level Design*, April 13, 1994.

[22] Rome Laboratory Technical Report RADC-TR-88-166, "Reconfigurable C2 DDP System", July 1988.

[23] J. D. Northcutt, R. Clark, D. Maynard, "Decentralized Real-Time Scheduling," Technical Report RADC-TR-90-182, Rome Laboratory, Rome, New York, August 1990.

*U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61094

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.